

# FINDING HAMILTONIAN CYCLES

A Thesis  
Presented to  
The Faculty of the Department of Computer Science  
Western Kentucky University  
Bowling Green, Kentucky

In Partial Fulfillment  
Of the Requirements for the Degree  
Master of Science

By  
Sridher Kaminani

August 2005

## FINDING HAMILTONIAN CYCLES

Date Recommended 5/24/2005

Dr. Uta Ziegler Uta Ziegler  
Director of Thesis

Dr. Claus Ernst Claus Ernst

Dr. Mustafa Atici Mustafa Atici

Dr. Elmer Gray Elmer Gray 8/8/05  
Dean, Graduate Studies and Research Date



## Acknowledgement

I would like to convey my sincere, heart felt thanks to Dr. Uta Ziegler and Dr. Claus Ernst. I feel extremely grateful and thankful for all that they have done to make this work possible. Their presence and guidance alone propelled my interest toward this work.

Words alone cannot describe my deep regards and gratitude to Dr. Ziegler for her constant motivation and invaluable support toward the completion of my thesis. Her immense participation and feedback, at various stages of the thesis, steered me to the completion of this work.

I would also like to thank Dr. Claus Ernst for his guidance in topics related to Knot Theory and his support at various stages of this work.

I am grateful to thank Dr. Mustafa Atici for his invaluable comments. I would like to thank my parents for their constant love, support and motivation.

Sridher Kaminani.

## Table of Contents

<b>ACKNOWLEDGEMENT .....</b>	<b>III</b>
<b>LIST OF FIGURES .....</b>	<b>VI</b>
<b>LIST OF TABLES.....</b>	<b>VIII</b>
<b>1. INTRODUCTION.....</b>	<b>1</b>
<b>2. BASIC GRAPH THEORY .....</b>	<b>3</b>
2.1 BASIC CONCEPTS OF GRAPH THEORY .....	3
2.2 FINDING A HAMILTONIAN CYCLE IN A GENERAL GRAPH IS HARD.....	8
2.3 BASIC THEOREMS TO RECOGNIZE THAT A GRAPH IS HAMILTONIAN OR NON-HAMILTONIAN .....	9
<b>3. HAMILTONIAN CYCLE ALGORITHMS .....</b>	<b>14</b>
3.1 GENERAL APPROACHES TO ALGORITHMS FINDING HAMILTONIAN CYCLES.....	14
3.1.1 <i>Backtrack Algorithms</i> .....	15
3.1.2 <i>Algorithms Using Heuristic Approach</i> .....	17
3.1.3 <i>Algorithms for Special Graphs</i> .....	18
3.2 BASIC IDEAS OF HAMILTONIAN CYCLE ALGORITHMS USING BACKTRACKING..	19
3.2.1 <i>Forced Paths</i> .....	19
3.2.2 <i>Pruning</i> .....	20
3.2.3 <i>Search Method</i> .....	25
3.2.4 <i>Vertex Selection</i> .....	26
3.3 BASIC IDEAS OF HAMILTONIAN CYCLE ALGORITHMS USING A HEURISTIC APPROACH .....	28
3.3.1 <i>Simple Extension</i> .....	29
3.3.2 <i>Cycle Extension</i> .....	30
3.3.3 <i>Rotation</i> .....	31
<b>4. DETAILED EXPLANATION OF VARIOUS HAMILTONIAN CYCLE ALGORITHMS USED.....</b>	<b>34</b>
4.1 VANDEGRIEND BACKTRACK ALGORITHM .....	34
4.2 SEMIHAM HEURISTIC ALGORITHM.....	38
4.3 BACKTRACK ALGORITHM WITH BRUTE-FORCE APPROACH .....	47
<b>5. EXPERIMENTS.....</b>	<b>49</b>
5.1 INPUT GRAPHS.....	49
5.2 EXPERIMENTAL METHODOLOGY .....	50

<b>5.3. RESULTS .....</b>	<b>50</b>
<i>5.3.1 Backtrack Algorithm using Brute-Force Approach.....</i>	<i>51</i>
<i>5.3.2 Vandegriend Backtrack Algorithm .....</i>	<i>53</i>
<i>5.3.3 SemiHam Heuristic Algorithm.....</i>	<i>55</i>
<i>5.3.4 All the three Algorithms.....</i>	<i>56</i>
<b>5.4. SUMMARY OF EXPERIMENTS .....</b>	<b>58</b>
<b>6. CONCLUSIONS.....</b>	<b>60</b>
<b>BIBLIOGRAPHY.....</b>	<b>61</b>

## List of Figures

Figure	Page No.
FIGURE 1: AN EXAMPLE GRAPH TO SHOW LOOP AND MULTIPLE EDGES. ....	4
FIGURE 2: A SAMPLE PLANAR GRAPH. ....	5
FIGURE 3: A SAMPLE 4-REGULAR PLANAR GRAPH ....	5
FIGURE 4: SMALLEST 4-REGULAR PLANAR GRAPH, WHICH IS NON-HAMILTONIAN ....	6
FIGURE 5: AN EXAMPLE TO ILLUSTRATE THEOREM 6 ....	12
FIGURE 6: OUTLINE OF THE BACKTRACK ALGORITHM USED TO FIND A HAMILTONIAN CYCLE. ....	16
FIGURE 7: OUTLINE OF THE HAMILTONIAN CYCLE ALGORITHM USING HEURISTIC APPROACH. ....	17
FIGURE 8: ILLUSTRATION OF A FORCED PATH ....	19
FIGURE 9: ILLUSTRATION OF THE REMOVAL OF EDGES AS A VERTEX IS ADDED TO THE PATH .....	22
FIGURE 10: ILLUSTRATION OF REMOVING THE EDGE CONNECTING THE END POINTS OF THE FORCED PATH. ....	22
FIGURE 11: SIMPLE EXTENSION OF PATH $P$ AT END POINT $v_R$ . ....	30
FIGURE 12: CYCLE EXTENSION OF PATH $P$ . ....	31
FIGURE 13: ROTATION OF PATH $P$ . ....	32
FIGURE 14: PSEUDO CODE OF VANDEGRIEND BACKTRACK ALGORITHM. ....	35
FIGURE 15: PSEUDO CODE OF CALCULATE_BACKTRACK_ALGORITHM PROCEDURE AFTER ADDING BACKTRACK OPERATION. ....	37
FIGURE 16: PSEUDO CODE OF SEMIHAM ALGORITHM ....	39
FIGURE 17: PSEUDO CODE OF PROCEDURE SEMIHAMSTAGE. ....	40
FIGURE 18: PSEUDO CODE OF PROCEDURE TRYEXTENDPATH. ....	41
FIGURE 19: PSEUDO CODE OF PROCEDURE CYCLE EXTEND ....	41
FIGURE 20: PSEUDO CODE FOR PROCEDURE DO_ROTATION ....	43
FIGURE 21: PSEUDO CODE OF PROCEDURES SEMIHAM_COMPLETECYCLE AND TEST_ROTATION. ....	44
FIGURE 22: SAMPLE GRAPH TO ILLUSTRATE THE ROTATED PATHS ....	45
FIGURE 23: BREADTH FIRST SEARCH TREE OF ROTATED PATHS WITH END POINTS 10, 1, 4, AND 7. ....	46
FIGURE 24: PSEUDO CODE OF BACKTRACK ALGORITHM WITH BRUTE-FORCE APPROACH ....	48
FIGURE 25: ILLUSTRATING THE % GRAPHS BACKTRACK ALGORITHM USING BRUTE-FORCE APPROACH FOUND HAMILTONIAN CYCLE. ....	52
FIGURE 26: TIME TAKEN BY BACKTRACK ALGORITHM USING BRUTE-FORCE APPROACH TO FIND HAMILTONIAN CYCLE IN THE HAM GRAPHS AND TREE GRAPHS IN SECS. ....	52
FIGURE 27: ILLUSTRATING THE % GRAPHS VANDEGRIEND BACKTRACK ALGORITHM FOUND HAMILTONIAN CYCLE. ....	54

FIGURE 28: AVERAGE TIME TAKEN BY VANDEGRIEND BACKTRACK ALGORITHM TO FIND HAMILTONIAN CYCLE IN HAM GRAPHS AND TREE GRAPHS.....	55
FIGURE 29: ILLUSTRATING THE % GRAPHS FOUND HAMILTONIAN BY SEMIHAM HEURISTIC ALGORITHM .....	57
FIGURE 30: TIME TAKEN BY SEMIHAM ALGORITHM TO FIND HAMILTONIAN CYCLE IN HAM GRAPHS AND TREE 3GRAPHS.....	57
FIGURE 31: ILLUSTRATING THE TIME TAKEN TO FIND HAMILTONIAN CYCLE BY ALL THREE ALGORITHMS FOR.....	58
HAM GRAPHS.....	58
FIGURE 32: ILLUSTRATING THE TIME TAKEN TO FIND HAMILTONIAN CYCLE BY ALL THREE ALGORITHMS FOR .....	58
TREE GRAPHS .....	58

## List of Tables

Table	Page No.
TABLE 1: TABLE OF ROTATED PATHS WITH END POINTS 10, 1, 4, AND 7 .....	46
TABLE 2. RESULTS OF BACKTRACK ALGORITHM USING BRUTE-FORCE APPROACH .....	51
TABLE 3: RESULTS OF VANDEGRIEND BACKTRACK ALGORITHM .....	54
TABLE 4: RESULTS OF SEMIHAM HEURISTIC ALGORITHM.....	56

# FINDING HAMILTONIAN CYCLES

Sridher Kaminani

August 2005.

72 Pages

Directed by: Dr. Uta Ziegler

Department of Computer Science

Western Kentucky University

Finding a Hamiltonian cycle in a graph is used for solving major problems in areas such as graph theory, computer networks, and algorithm design. In this thesis various approaches of Hamiltonian cycle algorithms such as backtrack algorithms and heuristic algorithms, their basic ideas, and their actual implementations are studied.

Three specific implementations are explained in detail and tested with randomly generated 4-regular planar graphs that are 2-connected and 4-edge connected. The results are analyzed and reported.

## 1. Introduction

The Hamiltonian cycle is named after the famous Irish mathematician Sir William Rowan Hamilton. The Hamiltonian cycle in a graph is a cycle that passes through all the vertices exactly once. The problem of finding a Hamiltonian cycle in a graph is called the Hamiltonian Cycle Problem.

The solution for many well-known Computer Science problems, such as the Knight's Tour Problem and the Traveling Salesman Problem, involves finding a Hamiltonian Cycle. The Hamiltonian Cycle Problem is not only an important graph theory problem but also has wide applications in areas such as Algorithm Design, Computer Networks (such as Broadcast Routing [14]), Telecommunication, Molecular Biology (such as DNA Analysis [17]), Genetics, Cryptography [15], Operations Research, etc.

Finding a Hamiltonian cycle in a given graph is a NP-Complete problem, i.e., no deterministic algorithm has been developed until present date, which can find a Hamiltonian cycle in a polynomial time. The NP-Complete (Non-Deterministic Complete) problems are the problems that are quickly solved by a non-deterministic machine, but are exponential difficult to solve on a deterministic machine, in which it has to try every possibility in the search space in search of a solution in the worst case.

In Chapter 2, the basic terms used in this thesis are introduced. The basic theorems from graph theory, that describe the necessary conditions for a Hamiltonian cycle to exist in a graph, and how these theorems can be used in finding a Hamiltonian cycle in 4-regular planar graph are described in this chapter.



In Chapter 3, the general approaches used in order to solve the Hamiltonian Cycle Problem are described. A general overview of backtrack algorithms and heuristic algorithms used to solve the Hamiltonian Cycle Problem are provided in this chapter. The basic ideas of these algorithms are described in this chapter. Algorithms that find a Hamiltonian cycle in some special graphs are also briefly mentioned.

In Chapter 4, the implementations of the Hamiltonian cycle algorithms using backtracking, such as Vandegriend Backtrack algorithm and backtrack algorithm with the brute force approach, and Hamiltonian cycle algorithms using heuristic approach, such as SemiHam heuristic algorithm are described in detail.

In Chapter 5, various experimental data collected using the above three algorithms are discussed. Firstly, the graphs that are used to test the above three algorithms and the experimental methodology are described. The results of the experiments performed on these three algorithms are explained and a summary of these experiments is provided in this chapter.

In Chapter 6, the conclusions that are arrived at in this thesis are described.

## 2. Basic Graph Theory

This chapter contains various concepts of graph theory that are used in this thesis. The definitions of various terms used are mentioned in Section 2.1. A brief description of the Hamiltonian Cycle Problem in the general graphs is described in Section 2.2. In Section 2.3 the basic theorems of graph theory, which are used to recognize whether a given graph is a Hamiltonian or non-Hamiltonian, are described.

### 2.1 Basic concepts of Graph Theory

The basic concepts of graph theory used in this thesis are:

**Graph:** A graph  $G$  is a collection of two sets  $(V, E)$ . The objects of  $V$  are called vertices, the objects of  $E$  are called edges. Each edge in  $E$  is associated with a pair in  $V \times V$  namely, the end points of edge  $e$ .

A vertex is an element of  $V$ . An edge is an element of  $E$ . Often the edge connecting vertex  $v$  to vertex  $u$  is denoted by  $vu$  or  $(v, u)$ .

**Degree of a vertex:** The degree of a vertex is the number of edges connected to it. The degree of a vertex  $v$  is represented by  $d(v)$ .

**Loop edge:** If there is an edge from a vertex  $v$  to the same vertex  $v$ , then it is called a loop edge. The edge  $vv$  is a loop edge. In Figure 1, the edge  $aa$  is a loop edge.

**Multiple edges:** If there is more than one edge from a vertex  $v$  to vertex  $u$ , then these edges are called a multiple edge. In Figure 1, vertex  $b$  and vertex  $c$  are connected by more than one edges. So the two edges  $(b, c)$  and  $(b, c)$  form a multiple edge.

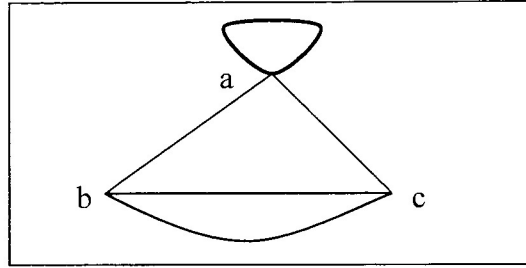


Figure 1: An example graph to show loop and multiple edges.

**Simple graph:** A graph that does not have any loop edges or multiple edges is called a simple graph.

**Trail:** A trail is a sequence of edges of the form  $v_1v_2, v_2v_3, v_3v_4, \dots, v_{n-1}v_n$ .

**Path:** A path is a trail  $v_1v_2, v_2v_3, v_3v_4, \dots, v_{n-1}v_n$  in which all vertices  $v_i$  are distinct.

The length of the path  $v_1v_2, v_2v_3, v_3v_4, \dots, v_{n-1}v_n$  is the number of edges it consists of; i.e.  $n-1$ . A path  $P$  of length  $k$  is represented as  $P(v_i, v_{i+1}, v_{i+2}, \dots, v_{i+k})$  for all  $j$  in

$0 \leq j \leq k-1$ ,  $v_{i+j}$  and  $v_{i+j+1}$  are connected by an edge  $v_{i+j}v_{i+j+1}$ .

**Hamiltonian path:** The Hamiltonian path of a graph is a path that visits each vertex in the graph exactly once. The length of a Hamiltonian path is  $|V| - 1$ , where  $|V|$  represents the number of vertices in the graph.

**Cycle:** A cycle is a closed path, a path which ends at the same vertex as it started from i.e. a cycle is a path  $v_1v_2, v_2v_3, v_3v_4, \dots, v_{n-1}v_n$  where  $v_1 = v_n$ .

**Hamiltonian cycle:** A Hamiltonian cycle is a closed Hamiltonian path. A graph containing a Hamiltonian cycle is known as Hamiltonian.

**Planar graph:** A planar graph is a graph that can be drawn on a plane, in such a way that the edges do not cross each other; i.e., edges intersect only at their common vertices. A sample planar graph is illustrated in Figure 2.

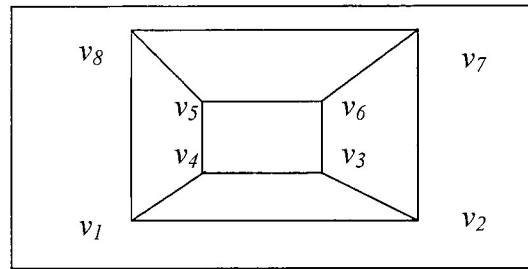


Figure 2: A sample planar graph.

**Regular graph:** A regular graph is a graph in which all the vertices have the same degree.

**K-Regular graph:** A graph is said to be k-regular if all the vertices have degree k.

A 0-regular graph consists of isolated vertices, a 1-regular graph consists of isolated edges, and a 2-regular graph is a cycle.

**4-Regular planar graph:** A planar graph, in which all the vertices have a degree 4, is known as 4-regular planar graph.

A sample 4-regular planar graph is illustrated in Figure 3.

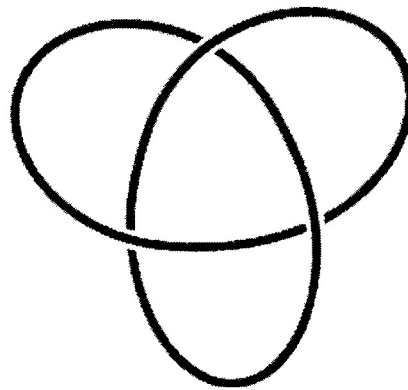
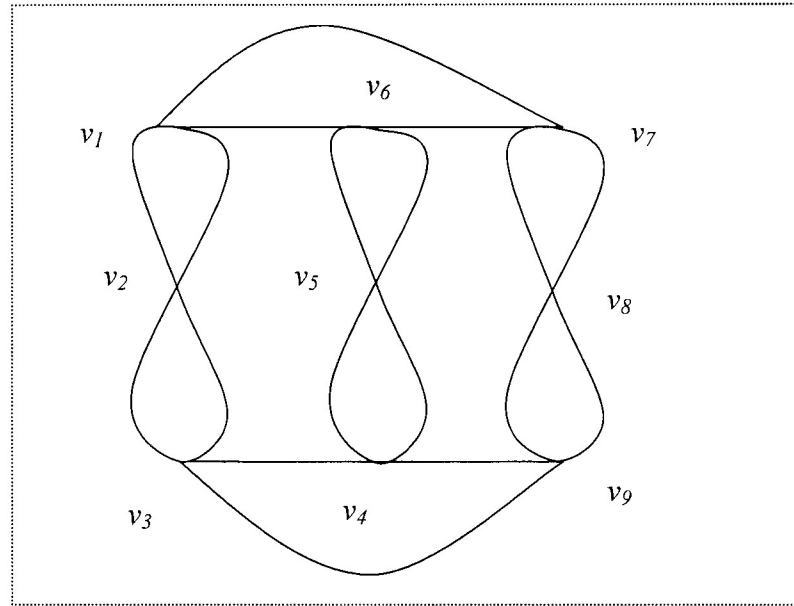


Figure 3: A sample 4-regular planar graph

The smallest 4-regular planar graph, which is not Hamiltonian, is illustrated in Figure 4.



**Figure 4: Smallest 4-regular planar graph, which is non-Hamiltonian**

**Connected graph:** A graph is said to be connected if there is a path from any vertex to any other vertex in the graph. A graph that is not connected is called a disconnected graph.

**Component:** Let a vertex  $v \in V$ , the component containing  $v$  consists of all the edges and vertices that can be reached by any path starting at  $v$ . The number of components of the graph  $G$  is represented by  $c(G)$ .

**Cut point or articulation vertex or cut vertex:**

A cut vertex of a connected graph is a vertex which if removed disconnects the graph and increases the number of components of the graph. A cut vertex can also be defined as a vertex  $v$  of a connected graph  $G$ , if and only if there exist vertices  $u$  and  $w \in V$ , where  $u \neq w \neq v$ , such that  $v$  is on every path from  $u$  to  $w$ .

**Cut set:** A set of vertices of a graph which if removed disconnects the graph. If  $S$  is a cut set of a connected graph  $G$ , then the graph  $G/S$  or  $(G-S)$ , which is obtained by the removal of the cut set, is disconnected. The size of cut set is represented by  $|S|$  is the number of vertices in the cut set..

**Bi-partite graph:** A bi-partite graph is a graph for which the set of vertices can be decomposed into two disjoint sets, such that no two graph vertices within the same set are connected by an edge.

**$k$ -connected graph:** A graph is  $k$ -connected if the deletion of any set of  $k-1$  vertices does not cause the graph to become disconnected.

**Biconnected graph:** A graph is a 2-connected or biconnected graph if the removal of one vertex does not disconnect the graph. It is a graph with no articulation vertex (cut vertex). All Hamiltonian graphs are biconnected.

**Complete graph:** A complete graph is a graph in which each pair of graph vertices is connected by an edge.

**Independent set of a graph:** An independent set of a graph is a set of vertices in which no two vertices are connected to each other by an edge.

**Neighborhood of a vertex  $v$ :** The set of vertices, which are adjacent to vertex  $v$ , i.e. connected to  $v$  by an edge, is known as the neighborhood of the vertex  $v$ . It is denoted by  $N(v)$ . The neighborhood of a set of vertices  $S$  in a graph  $G$ ,  $N(S)$ , is a set of vertices  $x_i$  such that  $(x_i, v)$  is an edge in  $G$ , for some  $v \in S$ .

**Closure of a graph:** The closure of a graph  $G$  is a graph obtained from  $G$  by recursively joining pairs of non-adjacent vertices  $v_1$  and  $v_2$ , where  $d(v_1) + d(v_2) \geq n$ , where  $n$  is the number of vertices in the graph  $G$ , until no such pair remains.

## 2.2 Finding a Hamiltonian Cycle in a General Graph is Hard

The problem of finding a Hamiltonian cycle in a graph is called the Hamiltonian Cycle Problem. The Hamiltonian Cycle Problem is one of the most famous problems in graph theory.

A problem is called NP (nondeterministic polynomial) if its solution (if one exists) can be guessed and verified in polynomial time by a nondeterministic machine; nondeterministic means that no particular rule is followed to make the guess.

If a problem is NP and all other NP problems are polynomial-time reducible to it, then the problem is NP-complete.

Thus, finding an efficient algorithm for any NP-complete problem implies that an efficient algorithm can be found for all such problems, since any problem belonging to this class can be transformed into any other member of the class in a polynomial time. It is not known whether any polynomial-time algorithms will ever be found for NP-complete problems, and determining whether these problems are tractable or intractable remains one of the most important questions in theoretical computer science.

To find a Hamiltonian cycle in a general graph is an NP-Complete problem and no deterministic polynomial-time algorithm has been discovered to find a Hamiltonian cycle in a general graph. The solution to many well-known Computer Science problems such as the Knight's Tour Problem and the Traveling Salesman Problem involves finding a Hamiltonian Cycle; see [1, 9].

### 2.3 Basic Theorems to Recognize that a Graph is Hamiltonian or Non-Hamiltonian

In this section some basic theorems from graph theory are stated that describe the necessary conditions for the existence or nonexistence of a Hamiltonian cycle in a graph. How these theorems are applicable to 4-regular planar graphs will also be discussed in this section. Most of these theorems can be found in any book on graph theory, for example see [2].

*Theorem 1:*

In a graph with a Hamiltonian cycle, the degree of each vertex must be greater than or equal to two. If a vertex has a degree two, then both edges incident to that vertex must be part of any Hamiltonian cycle.

*Theorem 2:*

If a vertex has three neighbors of degree two in a graph, then the graph cannot contain a Hamiltonian cycle.

*Theorem 3:*

If a vertex  $v$  has two neighbors  $a$  and  $b$  in the graph, which are both of degree two, then all edges  $(v, x)$ , where  $x \neq (a, b)$ , cannot be included in any Hamiltonian cycle of that graph.

*Theorem 4:*

If a path  $P(v_1, v_2, \dots, v_k)$  exists in a graph  $G$  with the length  $k - 1 < n - 1$  where  $n$  is the number of vertices in the graph, and  $d(v_2) = d(v_3) = \dots = d(v_{k-1}) = 2$  then the edge  $(v_1, v_k)$  cannot be in any Hamiltonian cycle.



A famous Theorem in graph theory is the following theorem published by Tutte in 1956 (see [11, 12]):

*Theorem 5:*

Every 4-connected planar graph has a Hamiltonian cycle.

*Observation 1:*

In a graph, if a vertex exists with degree one, then the graph cannot have a Hamiltonian cycle.

*How are these theorems useful for 4-regular planar graphs?*

All vertices in a 4-regular planar graph initially have a degree four. In order for all the above theorems to be used, the graph needs to have vertices of degree two, so they do not apply directly. However, after removing loop edges and all the edges making up a multiple edge except one in the graph, the graph may now have vertices of degree two, and it is at this point the above theorems can be applied. Deleting an edge that is either a loop or part of a multiple edge has no effect on whether or not a graph is a Hamiltonian. When all such edges are removed, the graph is a simple graph. When an edge is deleted in a graph in the search, it can possibly be a source of further edge deletions.

After removal of loop and multiple edges, if there is a vertex of degree one in the graph, then the graph cannot have a Hamiltonian cycle. If the graph contains degree two vertices then the edges incident to these vertices must be included in every Hamiltonian cycle for that graph (according to Theorem 1).

If the graph has vertices of degree two, a search can be made to count the neighbors of each vertex that have a degree two. If the graph has any vertex, which has

more than two vertices of degree two as neighbors, then the graph contains no Hamiltonian cycle (according to Theorem 2).

At any point if the graph contains any vertex  $v$ , which has two neighbors  $a, b$  of degree two, then the other edges of that vertex to the vertices other than  $a$  or  $b$  cannot be in the Hamiltonian cycle (according to Theorem 3) and can be deleted. If the graph contains a series of degree two vertices (this is called a forced path), and if the length of the path is less than the number of vertices in the graph, then the edge connecting the end points of the path cannot be in any Hamiltonian cycle of that graph (according to Theorem 4) and can be deleted.

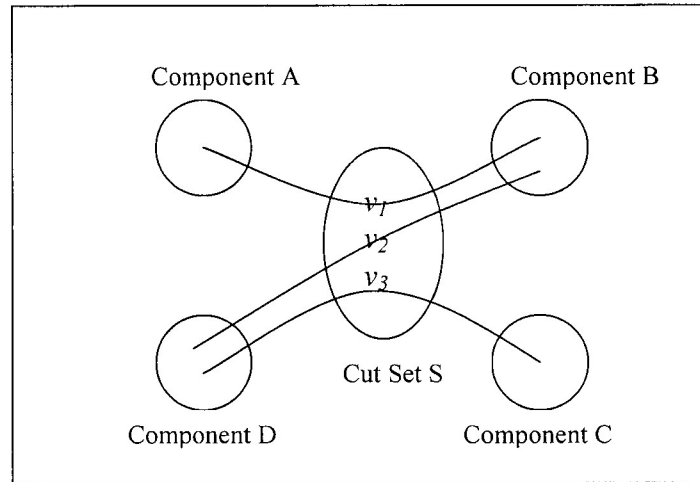
Theorem 5 says nothing about graphs that are not 4 connected planar graphs, i.e., the graphs could be Hamiltonian or non-Hamiltonian. It is possible to check whether the graph is Hamiltonian using this theorem by checking if the graph is planar and 4-connected. However this theorem does not tell us how to find a Hamiltonian cycle in the graph.

*Theorem 6:*

If  $S$  is a cut set of a graph  $G = (V, E)$ ,  $S \subseteq V$ , then no Hamiltonian cycle can exist when  $c(G/S) > |S|$  and no Hamiltonian path can exist when  $c(G/S) > |S| + 1$ , where  $c(G)$  is the number of components in the graph  $G$ .

If the number of components in the graph after the removal of the cut set  $S$  is greater than the size of the cut set, then we cannot have a path connecting all the components of the graph. In order to have a Hamiltonian cycle there needs to be at least  $c(G/S)$  number of vertices in the cut set. This scenario is illustrated in Figure 5.

In Figure 5, a graph  $G$  is given such that,  $G - \{v_1, v_2, v_3\}$  has four components A, B, C, and D. Any additional path from component C to component A, which is needed to close the cycle, requires a fourth vertex in the cut set. Thus no such path exists, no cycle can exist and  $G$  cannot be Hamiltonian.



**Figure 5: An example to illustrate Theorem 6**

*Corollary 6.1:*

If the graph  $G$  is bipartite with bipartition  $(X, Y)$  and  $|X| \neq |Y|$  then no Hamiltonian cycle exists.

*Note:* This Theorem is not useful for the 4-regular planar graphs, since almost all 4 regular planar graphs are not bipartite.

*Corollary 6.2:*

For any independent set  $S$  of graph  $G = (V, E)$  with neighborhood  $N(S)$ ,

If  $V/(S \cup N(S))$  is not an empty set, no Hamiltonian cycle can exist when

$$|N(S)| \leq |S|.$$

*Theorem 7:*

If the graph  $G = (V, E)$  is simple and vertices  $a, b \in V$  and  $(a, b) \notin E$  and  $d(a) + d(b) \geq n$  then  $G$  is Hamiltonian if and only if  $G' = (V, E')$  is Hamiltonian, where  $E' = E \cup \{(a, b)\}$ .

*Corollary 7.1:*

In a simple graph  $G (V, E)$  if  $|V| \geq 3$  and if  $\delta(G) \geq |V|/2$  then  $G$  is Hamiltonian, where  $\delta(G)$  represents the minimum degree of the graph.

*Theorem 8:*

A simple graph is Hamiltonian if and only if its closure is Hamiltonian.

*Corollary 8.1:*

A simple graph  $G$  with  $n \geq 3$  is Hamiltonian where  $n$  is the number of vertices in the graph, if the closure of graph  $G$  is complete.

*Theorem 9:*

A simple graph  $G$  has a non-decreasing degree sequence  $\{d_1, d_2, \dots, d_n\}$  with  $n \geq 3$ . If there is no  $m < n/2$  for which  $d_m \leq m$  and  $d_{n-m} < n - m$ , then  $G$  is Hamiltonian.

*How these theorems are useful for 4-regular planar graphs:*

Theorems 7 to 9 and their corollaries are valid for graphs that have vertices whose sum of degrees is larger than  $n$ . But the vertices in 4-regular planar graphs have a uniform degree of four. So, these theorems are not useful for 4-regular planar graphs.

### 3. Hamiltonian Cycle Algorithms

In this chapter an overview of various algorithms for solving the Hamiltonian Cycle Problem is presented. The various approaches used in order to solve the Hamiltonian Cycle Problem are outlined in Section 3.1. The overview of backtrack algorithms used to solve the Hamiltonian Cycle Problem is provided in Section 3.1.1, and the overview of algorithms that use a heuristic approach to solve the Hamiltonian Cycle Problem is given in Section 3.1.2. In Section 3.1.3, the description of algorithms that find Hamiltonian cycle for some special types of graphs is given. The basic ideas of backtrack algorithms that are used to solve the Hamiltonian Cycle Problem, such as forced paths, pruning, various vertex selection methods, and search methods are explained in Section 3.2. The basic ideas of algorithms using a heuristic approach for solving the Hamiltonian Cycle Problem such as simple extension, cycle extension and rotation are explained in Section 3.3.

#### 3.1 General Approaches to Algorithms Finding Hamiltonian Cycles

Finding a Hamiltonian cycle in a given graph is an NP-C problem. No algorithm can find a Hamiltonian cycle in deterministic polynomial time for all graphs.

The search space of a problem is the set of all possible solutions to the problem. The size of search space for Hamiltonian Cycle Problem is the factorial of the number of vertices of the given graph. The search space for finding a Hamiltonian cycle is the set of partial paths, which can be explored when searching for a Hamiltonian cycle. During the search, there are  $n$  vertices to select to find a Hamiltonian cycle; if the first vertex is

chosen then the remaining  $n - 1$  vertices can be selected in  $(n-1)!$  ways; that is the next vertex can be selected in  $n - 1$  ways, then the next vertex can be selected in  $n - 2$  ways, etc. There are  $n$  different options to select the first vertex. Thus the size of the search space for the Hamiltonian Cycle problem is the factorial of the size of the given graph.

In order to find the Hamiltonian cycle, the algorithms need to use either backtracking or some kind of heuristics. The algorithms that use backtracking always find the Hamiltonian cycle if the graph contains a Hamiltonian cycle. These backtrack algorithms search for every possible solution and backtrack if it cannot find one. The heuristic algorithms find the solution by reducing the search space by following certain heuristics that help the algorithm in finding the solution. There are some algorithms that find Hamiltonian cycles for some special types of graphs. They are described in Section 3.1.3.

### 3.1.1 Backtrack Algorithms

These algorithms will find the Hamiltonian cycles if the input graph has one, or will determine that no Hamiltonian cycle exists in the graph. It searches for all possible solutions in the graph looking for a Hamiltonian cycle. Efficient algorithms use known properties of the graphs with or without Hamiltonian cycle to restrict and guide the search.

In the worst case, these algorithms check every option in the search space to find a Hamiltonian cycle, and so they will not return until they find a Hamiltonian cycle or until it is clear that no Hamiltonian cycle exists. A time limit is used to stop the search after the time limit is reached.

The backtrack algorithm that is used to find the Hamiltonian cycle is recursive in nature. The pseudo code of these algorithms is illustrated in Figure 6.

```

Recursive_Backtrack(path  $P$ , end point  $e$ )
  If length ( $P$ ) =  $n-1$ 
    result = Change_to_cycle( $P$ ).
    if result = success then
      return success;
    else
      return failure;
  else
    for each unvisited neighbor  $x$  of  $e$ 
      Add  $x$  to path  $P$  to get  $P'$ 
      result = Recursive_Backtrack( $P'$ ,  $x$ )
      If (result) then
        Set  $P'$  to  $P$ ;
        return success;

    return failure;

```

**Figure 6: Outline of the backtrack algorithm used to find a Hamiltonian cycle.**

These algorithms find the solution by building up the path vertex by vertex, by calling the procedure Recursive\_Backtrack with the path built in the procedure so far as input. The number of active Recursive\_Backtrack function calls is equal to the *length of the current path* - 1. When the length of path reaches  $n - 1$  (Hamiltonian path), where  $n$  is the number of vertices in the graph, the algorithm tries to change the path into a cycle. If the algorithm successfully changes the path into a cycle then a Hamiltonian cycle is obtained.

The Recursive\_Backtrack procedure always tries to extend the path at the end point with one of the end point's unvisited neighbors. If this procedure is not able to extend the path, or the path obtained does not allow the Hamiltonian cycle to exist, then the procedure backtracks by removing the vertex added (either in the loop of the current

invocation of the Recursive\_Backtrack procedure or -- if all neighbors of the end point have been visited – in the loop of the prior invocation) and repeating the above process, by selecting a different unvisited neighbor of the end point.

### 3.1.2. Algorithms Using Heuristic Approach

The Hamiltonian cycle algorithms that use heuristic approach are fast, i.e., they run in linear or polynomial time. These algorithms work by using heuristics - general rules of thumb - to guide their search for a solution. These heuristics help the algorithm to execute quickly, but the algorithm cannot guarantee that it will find a solution even if one exists. The pseudo code of a basic Hamiltonian cycle algorithm that uses heuristic approach is illustrated in Figure 7.

```

Heuristic Algorithm (Graph  $G$ , Path  $P$ )
  For Stage 1 to  $n - 1$ 
    extended = Extend the path  $P$  by adding a new vertex to the path.
    If (!extended) then
      extended = transform the path and extend the path.
      If (!extended) then
        return failure;
  Stage  $n$ :
    result = complete the path  $P$  into cycle.
    If (result) then
      return success;
    else
      return failure;
  
```

Figure 7: Outline of the Hamiltonian cycle algorithm using heuristic approach.

The Hamiltonian cycle algorithms using a heuristic approach always try to extend the partial solution path, i.e., the length of the partial solution is always non-decreasing. No backtracking is ever done in these algorithms. Different Hamiltonian cycle algorithms using a heuristic approach use different heuristics to extend the path. The basic ideas of



these algorithms that are used to extend the path are discussed in Section 3.3. The actual implementation of the one such algorithm SemiHam is explained in Section 4.2.

Hamiltonian cycle algorithms using a heuristic approach work in stages. The input to stage  $i$  ( $i < n$ ) is the path of length  $i - 1$ ; if the stage is successful, a path of length  $i$  is produced. Stage 1 begins with a path containing an initial vertex. In the stage  $n$  the algorithm tries to close the path (Hamiltonian path) of length  $n - 1$  into a cycle (Hamiltonian cycle). If the  $n^{\text{th}}$  stage is successful, then a Hamiltonian cycle is found.

These algorithms always try to extend the path at one of its end points. If the path cannot be extended with the current end point, the path is transformed repeatedly into new paths with the same vertices as the old path, but the vertices occur in different orders and with at least one different end point until a path is found that can be extended. If the path formed contains all the vertices of the graph, then the algorithm tries to form a cycle. If the cycle can be formed successfully, then the algorithm returns success. If the cycle cannot be formed, then the algorithm returns failure.

### 3.1.3 Algorithms for Special Graphs

Mathematicians have developed various algorithms that find Hamiltonian cycles in different types of special graphs. These algorithms take advantage of the characteristics of the special graphs, which often allows for an approach that does not work on general graphs, but that is more efficient for these graphs.

Norishige Chiba and Takao Nishizeki [3] developed an algorithm that finds Hamiltonian cycles in 4-connected planar graphs in linear time. Gregory Gutin [6] developed an algorithm that finds Hamiltonian cycles in Quasi-transitive digraphs in polynomial time.



the edges  $v_j v_b$  or  $v_j v_a$  must not be included and the edge  $v_j v_{j+1}$  must be included in order to obtain a Hamiltonian cycle. After choosing vertex  $v_{j+1}$ , here is only one option to choose the next vertex i.e., vertex  $v_{j+2}$  (as  $d(v_{j+1}) = d(v_{j+2}) = 2$ ). So if vertex  $v_{j+2}$  is chosen, then for the next vertex there is no other option other than  $v_{j+3}$ , and so on until vertex  $v_k$  is reached. The path through  $(v_{j+1}, v_{j+2}, \dots, v_{j+k-1})$  has been forced, i.e., no choice was possible once  $v_j$  was reached. So this path is called a forced path.

While searching for a Hamiltonian cycle if a degree two vertex is encountered as a neighbor of the end point of the forced path then it should be added to the path. If two degree two vertices found as neighbors of the end point of the path, then the algorithm returns that a Hamiltonian cycle cannot exist in that graph,

### 3.2.2 Pruning

The operations that are used to reduce the size of the search space are called pruning operations. These operations are developed by making use of various theorems in graph theory that are necessary conditions for the existence or nonexistence of a Hamiltonian cycle in the graph (see Section 2.3) to reduce the size of the search space.

*Design Issues of Pruning operations:*

1. Since the algorithm spends time on processing the graph, other than directly finding a Hamiltonian cycle (pruning operation), the overall time required for finding a Hamiltonian cycle must be reduced after the inclusion of pruning.
2. The time, which is consumed in the execution of the pruning operations, must be appropriate to the reduction it makes to the search space.

Pruning can be done at different times during the search for a Hamiltonian cycle.

1. Initial pruning - It is performed once before the algorithm begins its search.
2. Search pruning - It is performed at each stage of the search.

Pruning is divided into two sub-categories.

1. *Graph reduction*: In this category, the algorithm checks the part of the graph to determine if an edge can be deleted.
2. *Global checking*: In this category, the algorithm checks the graph for certain properties, which provide conclusive evidence whether or not a Hamiltonian cycle exists in a graph.

Both initial pruning and search pruning contain graph reduction as well as global checking.

*Graph Reduction*:

In graph reduction the algorithm checks only a part of the graph to determine whether an edge can be deleted.

The various operations that can be performed for graph reduction are:

1. Modify the graph after each time a new edge and a new vertex has been added to the path (leave the edges of the initial vertex since they are needed in order to close the path).

In the graph  $G$ , shown in Figure 9, if the algorithm begins its search with  $v_1$  as initial vertex, and if the algorithm adds  $v_2$  and then  $v_3$  to the path, then the edges  $v_2v_7$ ,  $v_2v_8$ , and  $v_2v_9$  can be deleted, as these edges cannot be part of the Hamiltonian cycle which contains  $v_1$ ,  $v_2$  and  $v_3$  in this order (the path becomes a forced path). The algorithm leaves the edges of initial vertex as they are needed in order to close the cycle.

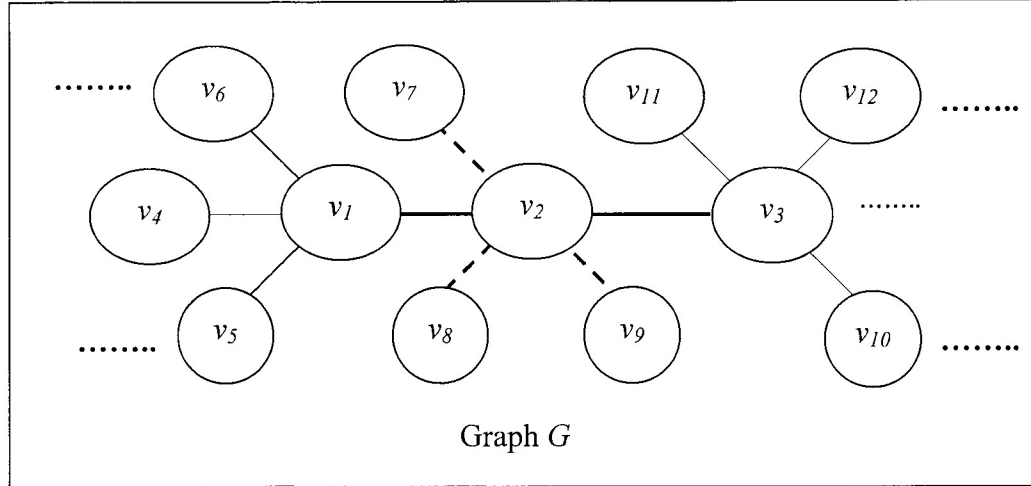


Figure 9: Illustration of the removal of edges as a vertex is added to the path

2. In a forced path  $P_f = (v_a, v_{a+1}, \dots, v_{a+k})$  with  $\text{length}(P_f) < n - 1$  if there exists an edge between the end points  $v_a$  and  $v_{a+k}$  then that edge can be pruned, i.e., the edge connecting the endpoints of the forced path can be deleted if the length of the forced path is less than  $n - 1$ . See Theorem 4 in Section 2.3.

In the Figure 10, the graph given consists of a forced path  $P_f$  with vertices  $v_a, v_{a+1}, \dots, v_{a+k}$ , and length of  $P_f < n - 1$  (i.e. there is at least one vertex of the graph which is not part of the forced path  $P_f$ ). The edge connecting the end points  $v_a$  and  $v_{a+k}$  is to be deleted, as the forced path becomes a cycle if the edge were to be added. If the length of the forced path is equal to  $n - 1$ , then the edge closes the forced path into a Hamiltonian cycle.

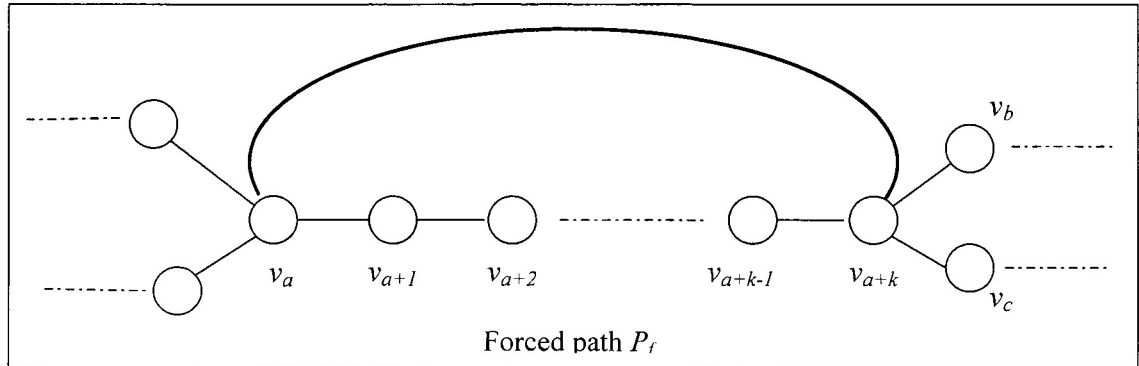


Figure 10: Illustration of removing the edge connecting the end points of the forced path

*Global Checking:*

Global checking evaluates the *entire graph* to check whether it possesses some property that ensures that there is no Hamiltonian cycle.

The various operations that can be performed for global checking are:

- a. The graph is checked for a vertex with degree one. No path can exist through a vertex with degree one and so no Hamiltonian cycle can exist. See Observation 1 in Section 2.3.
- b. The graph is checked for connectivity. If the graph is not connected then no Hamiltonian cycle can exist. See Theorem 1 in Section 2.3.
- c. The graph is checked for a cut vertex. If a graph contains a cut point no Hamiltonian cycle can exist. See Theorem 6 in Section 2.3.
- d. The graph is checked whether it is a bipartite graph with unequal partition sets. If this is the case for the given graph then no Hamiltonian cycle can exist. See Corollary 6.1 in Section 2.3.

The graph reduction involves modifications of the graph by deleting edges, so global checking needs to be done after all graph reduction operations are finished.

*Initial Pruning:*

Initial pruning is performed only once at the start of the search. In global checking the algorithm checks whether the graph has certain properties like biconnectivity, etc, which needs to be ensured in order for a Hamiltonian cycle to exist. Verifying whether the graph possesses such properties in the initial pruning may increase overall efficiency of the algorithm.

Testing for biconnectivity is the same as testing for a cut vertex, as biconnected graphs do not have any cut vertices. Thus testing for biconnectivity is a good way of eliminating non-Hamiltonian graphs; if there is a cut vertex in a graph then there is no Hamiltonian cycle. The biconnectivity pruning is an efficient pruning operation in this case, since the time required to find a cut vertex,  $\Theta(V+E)$  where  $V$  and  $E$  are the number of vertices and number of edges respectively (implemented in [13]) is much less than the time required to backtrack through the entire search space and to demonstrate that no Hamiltonian cycle exists. Frank and Martel [5] found that there are small random graphs of a particular edge density, in which very few biconnected graphs exist, that are non-Hamiltonian.

The success of initial pruning depends upon the structure and properties of the original graph. Most of the graph reduction operations require the existence of degree two vertices. Thus graph reduction operations in initial pruning fail in graphs that have a minimum degree of three.

#### *Search Pruning:*

Search pruning is performed in each step of the search. Its success depends upon the structure and properties of the current graph being searched. The structure of the original graph still affects the search pruning, but it is not as important as it was for initial pruning.

For graphs of mostly low degree vertices (graphs with degrees  $\leq 4$ ), graph reduction works well as the deletion of the edges quickly leads to the creation of new degree two or degree one vertices, which produces new forced paths and more edge deletions or produces the result that no Hamiltonian cycle exists in the graph. The process

of how an algorithm implements both graph reduction and global checking in initial pruning and in search pruning depends upon the algorithm used.

### 3.2.3 Search Method

After pruning the search space that remains must be searched in some systematic fashion. There are different methods to systematically move through the search space. They are as follows:

1. Single path method
2. Multi-path method
3. Double path method.

*Single path method:*

In this method a single path is maintained as the partial solution, and during the search, the path is extended from one end point. If the path cannot be extended then the algorithm backtracks.

*Multi-path method:*

In this method a list of paths is maintained that includes all forced paths. A path is selected at random, and the endpoint to extend the path from is randomly selected and the algorithm then attempts to extend this path. The extended path is added to the list. If no extension is possible to the selected path, the search method randomly picks another path.

*Double path method:*

In this method a single path is maintained, but it is extended from either end point. If the path cannot be extended, the algorithm backtracks.

The difference in performance of choosing one end point over the other end point depends on how the algorithm extends the path and when the algorithm implements the



pruning. The decision of selecting one end point over the other cannot be made ahead without knowing the structure of the entire graph.

Basil Vandegriend [13] explained that the multi-path method is not preferable over single path and double path methods as it is complex to implement, and the performance increase over the double path method is negligible. In a graph, if none of the paths can be extended, the multi-path method spends a lot of time in extending every path. After trying all the paths, it returns that no extension can be done.

### 3.2.4 Vertex Selection

Vertex selection determines how the algorithm selects the initial vertex and the next vertex at each step of the search.

There are two major heuristics that are used in vertex selection:

1. Low degree first heuristic
2. High degree first heuristic.

These heuristics explain how to select the next vertex to extend if there is more than one option to choose. Low degree first heuristic or high degree first heuristic does not select the lowest or highest degree vertex in the graph, but it selects the lowest or highest degree vertex from the subset of the vertices it can extend the path. If the end point of the path is incident to a forced edge, then that edge must be followed.

*Low degree first heuristic:*

In this heuristic, when more than one vertex is available to extend, the algorithm chooses the vertex with the lowest degree.

The low degree vertices are highly constrained. By first selecting the lower degree vertices, the algorithm leave the higher degree vertices for later. When there are fewer choices, there is a larger likelihood that the algorithm needs to backtrack. But since the later vertices have larger degree, there are more available options for forming a Hamiltonian cycle, thus improving the chances of the search succeeding.

*High degree first heuristic:*

In this heuristic, the algorithm selects at each step of the search, the vertex with highest degree when there is more than one option to choose. This maximizes the amount of pruning being done. Selecting the largest degree vertex among the neighbors of the end point and deleting other edges (other than the one we are following) leaves the low degree vertices for later. The edges deleted are incident on the low degree vertices, selecting them and removing the other edges, other than the one we are following further cause more edge deletions and further pruning, so the algorithm will soon run out of dead ends and backtrack.

The search method that is used (single path, multi-path or double path method) affects how we evaluate the vertex selection process. For multi-path and double path methods the end point to extend the path from can be selected by the low degree first or high degree first heuristic.

If the end point with the largest degree (among available vertices to extend the path) is selected, the number of edges pruned is maximized.

### 3.3 Basic Ideas of Hamiltonian Cycle Algorithms Using a Heuristic Approach

Hamiltonian cycle algorithms using a heuristic approach start with building a path by selecting an initial vertex and continue by extending the path from one of the current end points, one vertex at a time, until it reaches a path that uses each vertex of the graph exactly once (Hamiltonian path). It then completes the path by forming a cycle (Hamiltonian cycle).

The major limitations and strongholds of these algorithms are the techniques, i.e., heuristics they use in extending the path. These algorithms work by limiting the search space. This is what makes them efficient on the other hand, however this is what makes them fail. The algorithm needs to choose the best possible heuristic in order to find Hamiltonian cycles.

These algorithms work in stages. For each stage  $i$  for  $i = 1, 2, \dots, n - 1$  the input to the stage  $i$  is a path with  $i - 1$  edges, and the output, if the stage is executed successfully, is a path with  $i$  edges. The algorithm starts at stage one with one vertex and no edges. Then it proceeds to the next stages in sequence. If the algorithm is able to complete stage  $n - 1$ , the algorithm has found a Hamiltonian path in the graph. Then in stage  $n$ , the algorithm tries to close the path into a Hamilton cycle. If the final stage is successful the algorithm has found a Hamiltonian cycle.

#### Techniques Used for Extending the Path:

The Extension-Rotation technique is the basic concept used in many Hamiltonian cycle algorithms using a heuristic approach.

Let  $P = (v_l, \dots, v_r)$  be a path of length  $r - l + 1 < n$ , in the graph  $G$ , where  $n$  is the number of vertices in  $G$ .

There are three basic operations in this technique:

- a. Simple extension: This operation extends the path with one of the neighbors of one of the end points.  
(See Section 3.3.1.1.)
- b. Cycle extension: This operation can be used when the end points of the path are connected. The path is extended with one of the neighbors of one of the vertices of the path (which is not in the path).  
(See Section 3.3.1.2.)
- c. Rotation: This operation is used to change the end points of the path. It does not extend the path.  
(See Section 3.3.1.3.)

The first two operations, simple extension and cycle extension, are used in order to generate a path of length  $r$ . The last operation, rotation, is used if the algorithm is unable to perform a simple or cycle extension. It generates another path while keeping one end point fixed, which is of the same length as the original path, and uses the same vertices but in a different order.

### 3.3.1 Simple Extension

In simple extension, if there is an unvisited neighbor of an end point, then the algorithm includes it in the path (i.e., the algorithm extends the path) and the neighbor

added becomes a new end point of the path. The extension step starts with a path  $P = (v_1, v_2, \dots, v_r)$  of length  $r - 1$ . If  $v_k$  (where  $k=1$  or  $k=r$ , i.e.  $v_k$  is one of the end points of  $P$ ) has a neighbor  $w$ , with  $w \notin P$ , then the path  $P$  is extended by adding the edge  $(v_k, w)$  thus obtaining the path  $P' = (v_1, v_2, \dots, v_r, w)$  or  $P' = (w, v_1, v_2, \dots, v_r)$ . This scenario is shown in Figure 11. If there is more than one such neighbor of the end point  $v_k$ , the algorithm uses some heuristic in order to select one.

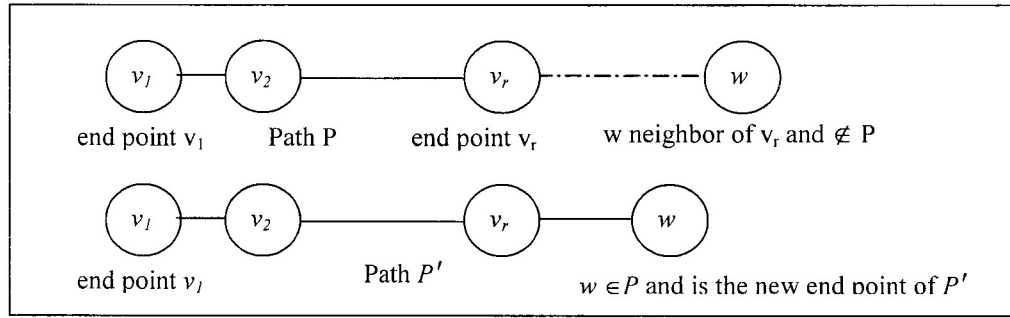


Figure 11: Simple Extension of path  $P$  at end point  $v_r$

### 3.3.2 Cycle Extension

Cycle extension can only be used if the end points of the current path are connected by an edge. The algorithm searches for a vertex in the path that has a neighbor and is not in the path. If the algorithm finds one, the algorithm performs a cycle extension as shown in the Figure 12. The added neighbor becomes the new end point.

The extension step starts with a path  $P = (v_1, v_2, \dots, v_r)$  of length  $r - 1$  with  $3 < r < n$ . If there is an edge  $(v_1, v_r)$  in the graph, i.e., the end points of the path are connected by an edge, adding this edge to the path  $P$  turns it into a cycle  $C$ , of length  $r$ . If the graph is connected, there must be vertex  $v_j$  in cycle  $C$ , which has a neighbor  $w$ ,  $w \notin C$ . Deleting the edge  $(v_j, v_{j+1})$  and adding the edge  $(v_j, w)$  results in a path  $P'$ .

$$P' = (w, v_j, v_{j-1}, \dots, v_1, v_r, v_{r-1}, \dots, v_{j+1}) \text{ of length } r.$$

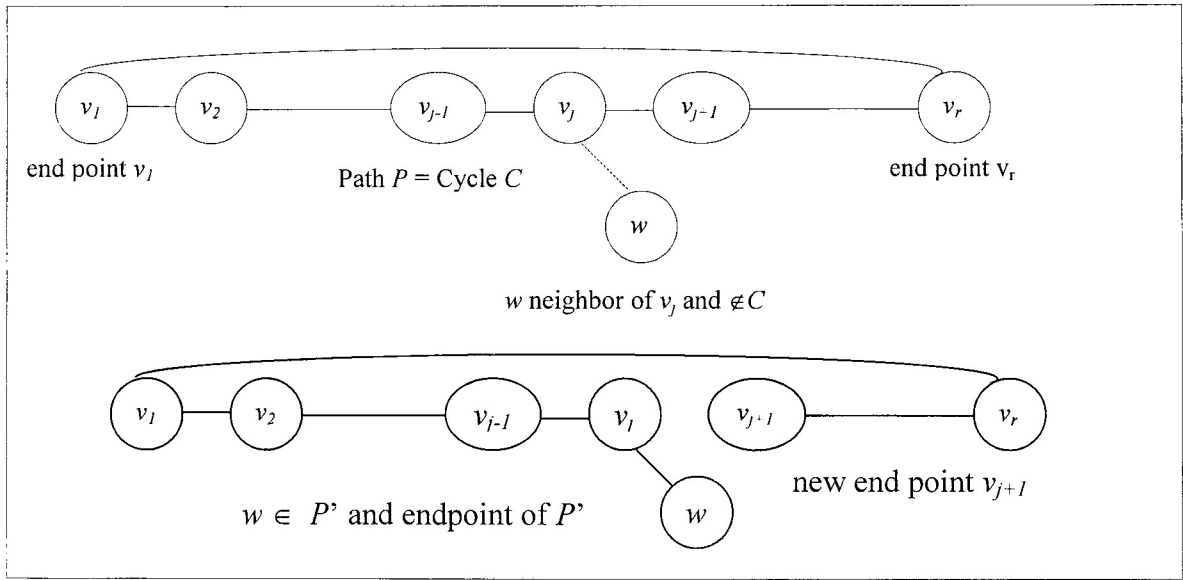


Figure 12: Cycle Extension of path  $P$

### 3.3.3 Rotation

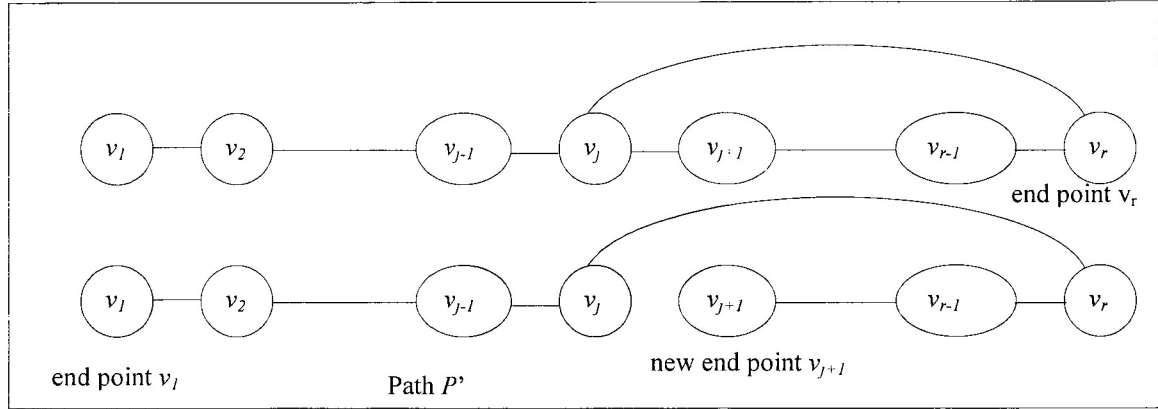
Rotation is used when simple extension and cycle extension are not possible.

In this operation, the algorithm searches for the neighbors of one of the end points (while the other end point is fixed), which are the elements of the current path, other than the end point's immediate neighbor. If such a neighbor is found, the algorithm performs rotation as shown in the Figure 13.

Rotation starts with a path  $P = (v_1, v_2, \dots, v_r)$  of length  $r - 1$ . Let  $N(v_r)$  be the set of all vertices which are neighbors of  $v_r$ , i.e., the set of vertices  $x_i$ , such that  $(x_i, v_r)$  is an edge in graph  $G$ . Let  $R(v_r)$  be  $N(v_r) \cap \{v_2, v_3, \dots, v_{r-2}\}$ . Every  $v_j \in R(v_r)$  is a neighbor of  $v_r$  already in the path. Pick one such  $v_j$ . Deleting the edge  $(v_j, v_{j+1})$  from  $P$  and adding the edge  $(v_j, v_r)$  to  $P$  results in a path  $P'$ .

$$P' = (v_1, \dots, v_j, v_r, v_{r-1}, \dots, v_{j+1}) \text{ of length } r - 1.$$

The rotation operation changes one of the end points of the current path while leaving the number of vertices as well as the actual vertices unchanged. It just rearranges the order of the vertices in the path. Rotation operations can also be applied while keeping  $v_r$  fixed and obtaining a path with a different starting point.



**Figure 13: Rotation of path  $P$**

At each stage of the algorithm, except the last stage, the algorithm either performs simple extension or a cycle extension, if this is possible, on the current path. If none of the extensions can be performed, the algorithm builds variations of the current path by applying rotations to the path and further rotations to the rotated paths until a path is found that can be extended with either simple or cycle extension. All the rotated paths have the same set of vertices as the original path, just in a different order. If the number of the rotated paths exceeds a certain threshold or if there are no more paths to rotate then the execution of the rotation operation in the current stage fails and the algorithm fails to find a Hamiltonian cycle.

In the last stage the algorithm tries to close the Hamiltonian path found into a Hamiltonian cycle. If the end points of the path are not connected, rotation operations are performed in order to change the end points until a pair of end points is found that are connected by an edge. After all possible rotations are performed, or the number of rotated

paths exceeds a threshold, and none of the rotated paths can be closed into a cycle, the algorithm fails to find Hamiltonian cycle.

Different Hamiltonian cycle algorithms using a heuristic approach use different heuristics to determine how many rotated paths are to be stored, which rotated paths are to be explored, and in which order. Even though no backtracking is used in these algorithms, these algorithms can successfully find Hamiltonian cycles in many graphs as the algorithms search most of the potential paths due to rotation operation (see Section 5.3.3 for the results of the SemiHam heuristic algorithm on 4-regular planar graphs).



## 4. Detailed Explanation of Various Hamiltonian Cycle Algorithms Used

This chapter introduces the various Hamiltonian cycle algorithms used and gives a detailed explanation about them. The three algorithms used are Vandegriend backtrack algorithm, SemiHam heuristic algorithm, and a backtrack algorithm using a brute-force approach. The Vandegriend backtrack algorithm developed by Basil Vandegriend [13] is explained in Section 4.1. The SemiHam heuristic algorithm developed by Gabriel Nivasch [10] is explained in Section 4.2. The backtrack algorithm that uses a brute-force approach to find a Hamiltonian cycle in a graph was developed by Richard Johnsonbaugh [7] is explained in Section 4.3.

### 4.1 Vandegriend Backtrack Algorithm

The Vandegriend backtrack algorithm solves the Hamiltonian Cycle Problem using backtracking. This algorithm uses the single path search method, graph reduction and global checking in initial and in search pruning (see Section 3.2.1). In this algorithm vertex selection uses the low-degree-first heuristic (see Section 3.2.4), and the initial vertex is selected randomly. The pseudo code for the Vandegriend backtrack method is given in Figure 14.

The Vandegriend backtrack algorithm implements the single path search method, i.e., it maintains only one partial solution path  $P$ , and the algorithm extends the path at a single end point. It selects the initial vertex randomly and adds the selected initial vertex to the path  $P$ , resulting in path  $P'$ . It then performs graph reduction on the graph  $G$  to get a graph  $G'$ , followed by global checking of  $G'$ . If the global checking indicates that no Hamiltonian cycle exists in  $G'$  with the path  $P'$ , then the algorithm selects a different

vertex and repeats the above process. Otherwise it calls recursively the procedure Calculate\_Backtrack\_Algorithm with the pruned graph  $G'$  and the updated path  $P'$  as input. If the path contains all the vertices of the graph (Hamiltonian path), then the Calculate\_Backtrack\_Algorithm tries to close the path into a Hamiltonian cycle. If the path can be successfully turned into a cycle then the algorithm returns “HC\_FOUND” and the Hamiltonian cycle in  $P$ . If the algorithm fails to turn the Hamiltonian path into a Hamiltonian cycle, then the algorithm returns that Hamiltonian cycle cannot exist in the graph.

*Input:* Current graph  $G$  which resulted from earlier pruning operations and path  $P$  in which each vertex along the path has degree two except the first and last vertices i.e. end points of the path.

*Output:* If a Hamiltonian cycle is found it returns HC\_FOUND, if no Hamiltonian cycle can exist in graph  $G$  with the path  $P$  it returns HC\_NOT\_EXIST, else it returns HC\_NOT\_FOUND.

**Calculate\_Backtrack\_Algorithm** (graph  $G$ , path  $P$ )

  If length( $P$ ) =  $n - 1$  then

    result = Change to Cycle ( $P$ );

    if (result) then

      return HC\_FOUND;

    else

      return HC\_NOT\_EXIST

  else

    For each eligible neighbor  $x$  of the end point  $e$  of the path  $P$  do

      Add  $x$  to path  $P$  as next vertex to get path  $P'$ .

      Apply graph reduction on the graph  $G$  to get the graph  $G'$ .

      If (global checking( $G'$ ) != HC\_NOT\_EXIST) then

        result = Calculate\_Backtrack\_Algorithm ( $G'$ ,  $P'$ );

        if (result = HC\_FOUND)

          Copy  $P'$  to  $P$ .

          return HC\_FOUND;

      return HC\_NOT\_FOUND;

End.

**Figure 14: Pseudo code of Vandegriend Backtrack Algorithm**

As the backtrack algorithms used to find a Hamiltonian cycle, in a given graph, do not return until they find a solution or have determined that no solution can be found, a time limit is used. The time limit is checked before the algorithm calls `Calculate_Backtrack_Algorithm` procedure. If the time limit has expired, the algorithm stops the search and returns that the Hamiltonian cycle is not found in the given graph.

The Vandegriend backtrack algorithm extends the path with one of the neighbors of the one end point while keeping the other end point fixed. If it finds a neighbor with degree two, it then takes the degree two vertex and then adds it to the path. If it finds two degree two vertices as neighbors to the end point, then the algorithm returns that a Hamiltonian cycle cannot exist in the given graph (See Section 3.2.1). If no neighbor of degree two exists and there is more than one eligible neighbor, then the algorithm selects the next vertex to extend by using the low-degree-first heuristic; that is the algorithm takes the vertex with the lowest degree. The detailed explanation about this heuristic is given in Section 3.2.4. If the path cannot be extended, the algorithm removes the vertex added to the path and backtracks.

The edges removed during pruning need to be restored as part of backtracking. An edge stack is maintained in the `Calculate_Backtrack_Algorithm` procedure to store the edges deleted during pruning. When the algorithm extends the path with one of the neighbors of the end point, the algorithm removes the edges connecting the end point and the end point's neighbor (not in the path) from the graph, and stores them in the edge stack. If backtracking is necessary the algorithm adds the edges deleted from the graph (that are stored in edge stack) back to the graph. The position of the edge stack is constantly monitored to ensure that the edges removed from the graph are added back

into the graph. This operation is added in the Calculate\_Backtrack\_Algorithm procedure and is illustrated in Figure 15.

*Input:* Current graph  $G$  which resulted from earlier pruning operations and path  $P$  in which each vertex along the path has degree two except the first and last vertices i.e. end points of the path.

*Output:* If a Hamiltonian cycle is found it returns HC\_FOUND, if no Hamiltonian cycle can Exist in graph  $G$  with the path  $P$  it returns HC\_NOT\_EXIST, else it returns HC\_NOT\_FOUND.

**Calculate\_Backtrack\_Algorithm** (graph  $G$ , path  $P$ )

```

If length ( $P$ ) =  $n - 1$  then
    result = Change to Cycle ( $P$ );
    if (result) then
        return HC_FOUND;
    else
        return HC_NOT_EXIST
else
    For each eligible neighbor  $x$  of the end point  $e$  of the path  $P$  do
        Add  $x$  to path  $P$  as next vertex to get path  $P'$ .

        Apply graph reduction on the graph  $G$  to get the graph  $G'$  and store the edges
        deleted in the edge stack.

        If (global checking( $G'$ ) != HC_NOT_EXIST) then
            If time limit has expired then return HC_NOT_FOUND.

            result = Calculate_Backtrack_Algorithm ( $G'$ ,  $P'$ );
            if (result = HC_FOUND)
                Copy  $P'$  to  $P$ .
                return HC_FOUND;
            else
                Add the edges deleted from the graph in the earlier step back into the graph from
                edge stack.
                Select a different neighbor and continue.

    return HC_NOT_EXIST;
End.
```

Figure 15: Pseudo code of Calculate\_Backtrack\_Algorithm procedure after adding backtrack operation

The Vandegriend backtrack algorithm maintains a *nodecount* variable in order to count the number of times the algorithm is trying to extend the current path. If the value

in *nodecount* exceeds a threshold value *MaxNodes* (which is equal to *Restart\_Increment* times the size of the graph initially, where *Restart\_Increment* is specified by the user), the algorithm restarts the search by selecting a new initial vertex, extending the path again from the starting, and increasing the *MaxNodes* value by the product of current *MaxNodes* value and *Restart\_Increment*. This operation is provided in order to ensure a certain randomness in the search.

#### 4.2 SemiHam Heuristic Algorithm

The SemiHam algorithm uses the heuristic approach to guide the search. No randomness is used in this algorithm. This algorithm uses the double path search method. Whenever one of the several eligible vertices needs to be selected, the algorithm takes the vertex with the smallest vertex number. The general structure of the heuristic algorithms used to find a Hamiltonian cycle in the given graph is specified in Section 3.1.2. The basic ideas of these algorithms are specified in Section 3.3. The pseudo code of the SemiHam algorithm is illustrated in Figure 16.

The SemiHam algorithm works in stages. The input to stage  $i$  is a path of length  $i-1$ , and if the stage executes successfully, the algorithm produces a path of length  $i$ . If the  $n-1^{st}$  stage is successful, then a Hamiltonian path is obtained. In the  $n^{th}$  stage, which is the final stage, the algorithm tries to close the Hamiltonian path into a Hamiltonian cycle. If this final stage is successful, then the algorithm found a Hamiltonian cycle.

The SemiHam algorithm starts by selecting the vertex with vertex number 0 as the initial vertex and adds it to the partial solution path. It then tries to extend the path by one edge  $n-1$  times, by calling the procedure SemiHamStage. If all the  $n-1$  stages are executed

successfully, then the algorithm calls procedure `SemiHam_CompleteCycle`. If this procedure is executed successfully, then the algorithm returns success and Hamiltonian cycle it has found.

Input: Graph  $G(V, E)$ ,  $n = |V|$  and the partial solution *Path* (empty path).  
 Output: If Hamiltonian cycle is found it returns success and Hamiltonian cycle in *Path*, else it returns failure

**SemiHam** (graph  $G$ , partial solution *Path*)  
 Take 0<sup>th</sup> vertex (first vertex) as initial vertex and add it to *Path*.  
 For  $i = 1$  to  $n-1$  do  
      $result = \text{SemiHamStage}(Path)$ .  
     If ( $result = failure$ ) then  
         return *failure*.  
 $result = \text{SemiHam\_CompleteCycle}(Path)$ .  
 return  $result$ .

Figure 16: Pseudo code of SemiHam Algorithm

*SemiHamStage*:

This procedure performs path extensions by adding one vertex to the path. The algorithm calls the procedure `TryExtendPath` with the partial solution path. This procedure tries to extend the path by one edge. If the path cannot be extended directly, then the algorithm calls the procedure `Do_Rotation` with the partial solution path and boolean value *false* (which indicates the algorithm is not trying to close the path), to change the end points of the path in search of a path that can be extended. The pseudo code of this procedure is shown in Figure 17.

*TryExtendPath*:

The `TryExtendPath` procedure is executed in order to extend the path without using rotations. In this procedure the algorithm first tries to extend the path, at one or the other end point, with its smallest unvisited neighbor. If the path can be extended the

algorithm returns success. This operation is called simple extension. The detailed operation of simple extension is explained in Section 3.3.1.

Input: Current partial solution *Path*.  
 Output: If the *Path* can be extended it returns the extended path *Path* else it returns failure.

**SemiHamStage** (*Path*)  
   result = TryExtendPath(*Path*).  
   if (result = success) then  
     return *success*.  
   else return Do\_Rotation (*Path*, *false*).

**Figure 17: Pseudo code of procedure SemiHamStage**

If simple extension operation fails and if the end points of the path are connected by an edge, then the algorithm uses the Cycle\_Extend procedure to apply a cycle extension. If this procedure returns success, then the algorithm returns success to the calling procedure, and returns the extended path; otherwise it returns failure to the calling procedure. The pseudo code of this procedure is shown in Figure 18.

Input: Current partial solution *Path* ( $v_1, v_2, \dots, v_r$ ), where  $r < n$  with endpoints  $v_1$  and  $v_r$ .  
 Output: If the *Path* can be extended it returns the extended path *Path* else it returns failure.

**TryExtendPath** (*Path*)  
   Find the eligible neighbors  $\Gamma(v_1)$  which are not in *Path*  
   If  $\Gamma(v_1)$  is not empty  
     Select a vertex  $x$  from  $\Gamma(v_1)$   
     Extend the *Path* at the end point  $v_1$  to  $x$ .  
     return success.  
   else  
     Find the eligible neighbors  $\Gamma(v_r)$  which are not in *Path*  
     If  $\Gamma(v_r)$  is not empty  
       Select a vertex  $x$  from  $\Gamma(v_r)$   
       Extend the *Path* at the end point  $v_r$  to  $x$ .  
       return success.  
   If  $(v_1, v_r)$  is an element of  $E$  then  
     return Cycle\_Extend(*Path*);  
   else  
     return failure;

**Figure 18: Pseudo code of procedure TryExtendPath**

*Cycle\_Extend:*

The Cycle\_Extend procedure is used to extend the path when the end points of the path are connected by an edge and when simple extension fails. When the end points of the path are connected by an edge, the path turns into a cycle. In order to perform an extension of the path, the path should not be a cycle. So in order to eliminate the cycle, a vertex in the path is searched that has an unvisited neighbor. If such a vertex is found to exist, then this procedure builds a new path and returns success, otherwise it returns failure. This operation is called cycle extension. The detailed operation of cycle extension is explained in Section 3.3.2. The pseudo code of this procedure is shown in Figure 19.

Input: Current partial solution *Path* ( $v_1, v_2, \dots, v_r$ ), where  $r < n$  with endpoints  $v_1$  and  $v_r$ .  
 Output: If the *Path* can be extended it returns the extended path *Path* else it returns failure.

**Cycle\_Extend** (*Path*)  
 Search for a vertex  $v_j$  in partial solution *Path*, which has an unvisited neighbor  $w$ .  
 If such a vertex  $w$  is not found then return failure.  
 Delete the edge  $(v_j, v_{j+1})$  from *Path* and add the edges  $(v_j, w)$  and  $(v_1, v_r)$  to the *Path*, which lead to a new path from  $w$  to  $v_{j+1}$ .  
 Return success.

**Figure 19: Pseudo code of procedure Cycle Extend**

*Do\_Rotation:*

The Do\_Rotation procedure is used when the end points of the current path do not allow the algorithm to continue with the next step. This procedure implements the rotation operation (see Section 3.3.3) to generate other alternative paths in the graph that use the same vertices, but in different orders. This procedure calls the Test\_Rotation procedure with the new rotated path and a boolean value “*justclosecycle*” which indicates



whether the algorithm is trying to extend the path or close the path. It tests whether the path can be extended or the end points of the path are connected. The `Do_Rotation` procedure generates rotated paths until a path is obtained, which when tested with `Test_Rotation` returns success. The `Do_Rotation` procedure uses a `pathList` to store the rotated paths. The original partial solution path is stored as the first element of the `pathList`. Each path from the `pathList` is taken and the rotation operation is performed to change the end point. The new path is checked to determine whether a path already exists in `pathList`, with the same end points as the new path; if so, the algorithm generates a different path. The checking operation is used to restrict the rotations on the path that must be stored.

The new path is checked by calling the procedure `Test_Rotation`. If the procedure `Test_Rotation` returns success, then the procedure `Do_Rotation` returns success. Otherwise the procedure `Do_Rotation` adds the new path to the end of the `pathList`, and again repeats the same process by taking the next path from the beginning of the `pathList`. The pseudo code of this procedure is shown in Figure 20.

The procedure `Do_Rotation` stops when the number of paths in the *pathList* reaches a threshold value of  $n^2$ , where  $n$  is the number of vertices of the graph. This restriction is used in order to ensure that the tree built in *pathList* is of polynomial size, and that the `SemiHam` algorithm terminates after the number of rotated paths reach this threshold value.

*SemiHam\_CompleteCycle:*

This procedure is used to close the Hamiltonian path into a Hamiltonian cycle. This procedure first checks whether the end points of the current path are connected. If

the end points are not connected, then the algorithm calls the procedure `Do_Rotation` with the Hamiltonian path, and a boolean value *true* that indicates the algorithm is trying to close the path into a cycle. The pseudo code of this procedure is shown in Figure 21.

```

Input: Current partial solution Path ( $v_1, v_2, \dots, v_r$ ), where  $r < n$  with endpoints
        $v_1$  and  $v_r$  and a boolean value.
Output: If the Path can be extended it returns the extended path Path else it
       returns failure.

Do_Rotation (Path, boolean justclosecycle)
Let pathList be the list of paths, which is used to store rotated paths.
Store the current path Path in the pathList
While pathList is not empty. do
    Take next path P ( $v_1, v_2, \dots, v_r$ ) with endpoints  $v_1$  and  $v_r$  from pathList.
    For  $v_j$  in  $\{v_1, v_r\}$  do
        Find  $I(v_j)$ , the set of vertices in P which are connected to  $v_j$ .
        Remove the immediate neighbors of  $v_j$  from  $I(v_j)$ .
        For each vertex  $w$  in  $I(v_j)$  do
            Let  $u$  be the vertex on P adjacent to  $w$  when moving away from  $v_j$ .
            If the pathList already contains a path with end points  $u$  and  $v_j$  then
                Continue;
            Rotate P to new path P' with vertices  $u$  and  $v_j$  as end points.
            If (Test_Rotation (P', justclosecycle) = success) then
                return success;
            else
                Add path P' to the pathList.
    return failure;

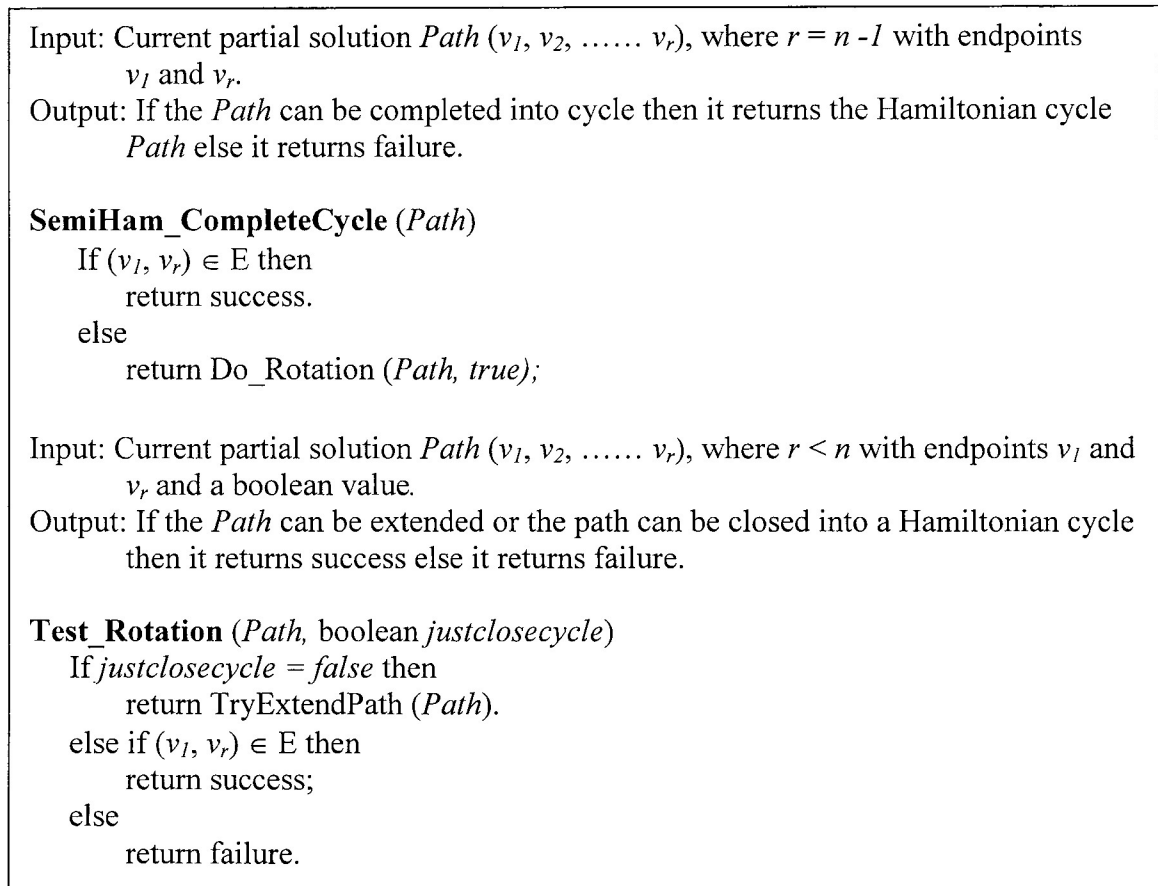
```

**Figure 20: Pseudo code for procedure `Do_Rotation`**

*Test\_Rotation*:

This procedure is used to find whether the rotated path can be extended or can be closed into a Hamiltonian cycle. It checks the value in the variable *justclosecycle*. If the value in *justclosecycle* is true, then it checks whether the end points of the path are connected by an edge; if so then it returns success.

If the value in *justclosecycle* is false, then it calls TryExtendPath to check whether the path can be extended, with either simple extension or cycle extension. The pseudo code of this procedure is shown in Figure 21.

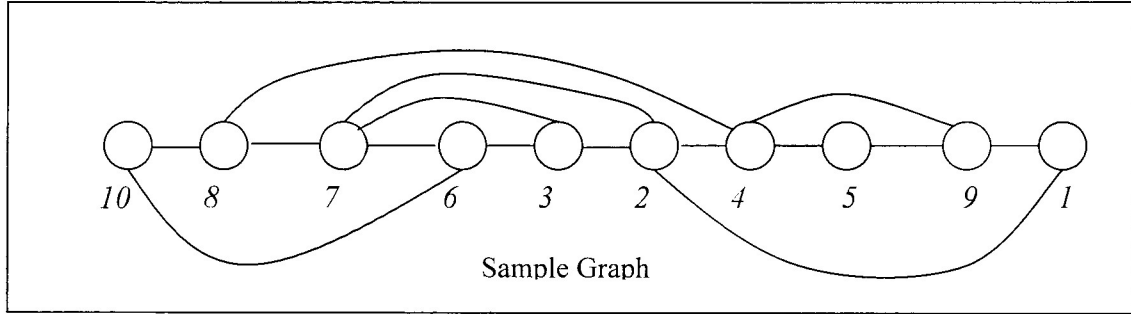


**Figure 21: Pseudo code of procedures SemiHam\_CompleteCycle and Test\_Rotation**

*How SemiHam generates the new rotated paths:*

The SemiHam algorithm builds a tree of rotated paths in a breadth first manner. It first generates all possible rotations available for the path, and checks each of them, whether they can be extended (or closed). If none of the paths can be extended (or closed), then it picks the first rotated path and performs all possible rotations for that

path. So rotated paths will be generated in a breadth first manner, this is illustrated with an example graph shown in Figure 22 in the Table 1.



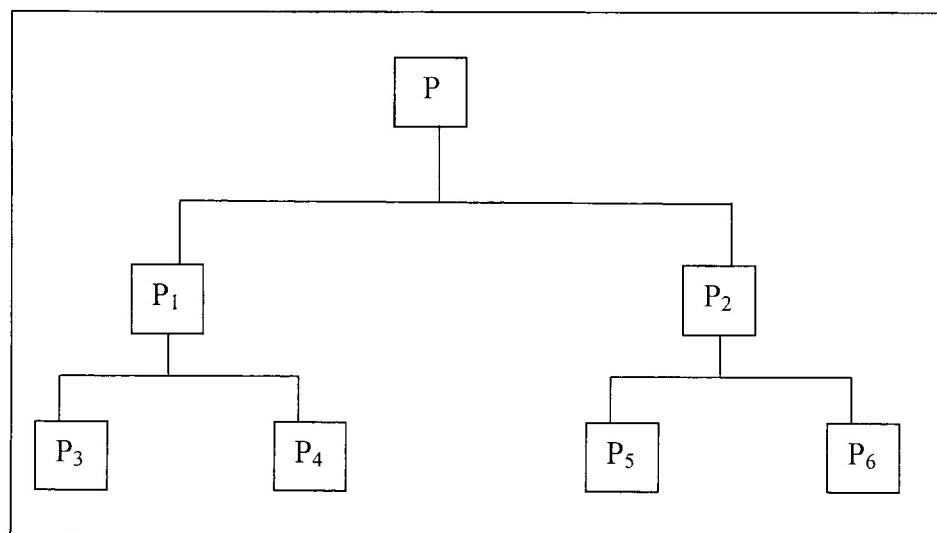
**Figure 22: Sample Graph to illustrate the rotated paths**

Let the path  $P$  be  $(10, 8, 7, 6, 3, 2, 4, 5, 9, 1)$  with end points 10 and 1. Let the end point 10 be connected with 8 and 6, end point 1 be connected with 9 and 2, and let vertex 7 be connected with 8, 6, 3, and 2 and vertex 4 be connected with 2, 5, 9 and 8. Then the SemiHam first generates all the rotations for end point 10, then for end point 1, and stores them in pathList. If any path produces an extension, then the procedure returns success. If none of the paths produce extension, then the procedure will take the first rotated path  $P_1$ , and apply rotations on either of the end points and so on. If the path  $P(v_1, v_2, v_3, \dots, v_j, v_{j+1}, \dots, v_k)$  is rotated with an end point  $v_k$  and if  $v_k$  has one of the neighbor  $v_j$ , then the path obtained after rotation is path  $P'(v_1, v_2, v_3, \dots, v_j, v_k, v_{k-1}, v_{k-2}, \dots, v_{j+1})$  is obtained. See Section 3.3.3 for more information. The rotated paths, which are generated for end points 10, 1, 7, and 4 are illustrated in Table 1. The tree of rotated paths built in breadth first manner is illustrated in the Figure 23.

**Table 1: Table of rotated paths with end points 10, 1, 4, and 7**

	Path	End point $v_k$	Neighbor of end point $v_k$ ( $v_j$ )	New end point of rotated path ( $v_{j+1}$ )
P	(10,8,7,6,3,2,4,5,9,1)			
P <sub>1</sub>	(7,8,10,6,3,2,4,5,9,1)	10	6	7
P <sub>2</sub>	(4,5,9,1,2,3,6,7,8,10)	1	2	4
P <sub>3</sub>	(6,10,8,7,3,2,4,5,9,1)	7	3	6
P <sub>4</sub>	(3,6,10,8,7,2,4,5,9,1)	7	2	3
P <sub>5</sub>	(9,5,4,1,2,3,6,7,8,10)	4	9	5
P <sub>6</sub>	(7,6,3,2,1,9,5,4,8,10)	4	8	7

The heuristics employed in the SemiHam algorithm are maximum size of the tree (*pathList*), and the restrictions on the rotated paths generated, i.e., only one rotated path with the same pair of end points is allowed. These restrictions are used in order to ensure that the tree built is of polynomial size, and that the SemiHam algorithm terminates at the latest after the number of rotated paths reaches this threshold value.

**Figure 23: Breadth first search tree of rotated paths with end points 10, 1, 4, and 7**

### 4.3 Backtrack Algorithm with Brute-Force Approach

This backtrack algorithm uses the brute-force approach to find Hamiltonian cycle in the graph. It uses the single path search method, i.e., it maintains a single partial solution path and tries to extend it at one end point. It takes 0<sup>th</sup> vertex, i.e., first vertex as initial vertex. This algorithm uses a different structure than the one specified in Section 3.1.1. The pseudo code of this method is illustrated in Figure 24.

This backtrack algorithm starts with a path of length zero with the initial vertex as the only vertex in the partial solution path. It then builds the path vertex by vertex until a path of length,  $n-2$  where  $n$  is the number of vertices of the graph, is obtained. It then looks for last vertex, which is both connected to the end point of the current path and starting point of the path. If such a vertex is found, the path built, a Hamiltonian cycle, is obtained and the algorithm returns success.

This algorithm uses a brute-force approach to select the next vertex, i.e. it takes every vertex in the graph and checks whether it is already a part of the partial solution, if not it checks whether it is connected to end point of the path. If such a vertex is found the algorithm takes it into partial solution, and searches for the next vertex. This algorithm backtracks when no such vertex is found. It backtracks by changing the end point of the partial solution by trying all the vertices in the graph.

As the backtracking algorithms used to find a Hamiltonian cycle do not return until the algorithm finds the solution, or it can be determined that a Hamiltonian cycle cannot exist in the graph, and takes an indefinite time to complete in the worst case, a time limit is used. This algorithm returns failure after the time limit has expired.

*Input:* Graph  $G (V, E)$ ,  $n = |V|$  and the partial solution be Path.

*Output:* If Hamiltonian cycle is found it returns success and the cycle in the Path,  
else it returns failure

**Bruteforce\_Hamiltonian** (Graph  $G$ )

Select first vertex as initial vertex.

Add the initial vertex to the Path.

do

- a) Start from the first vertex and check every vertex in the graph, whether it belongs to path and it is connected to the end point of Path. If so add the vertex to the Path, else try the next vertex until all the vertices of the graph have been tried.
- b) If no such vertex is found, then change the end point of the Path by trying every vertex in the graph.
- c) If all the vertices have been tried and no such vertex is found with which the path can be extended then return failure.
- d) If a Path of length  $n-1$  is found then it checks whether the end points are connected, if connected return success.

Until a closed path of length  $n$  is found then return success

**Figure 24: Pseudo code of Backtrack Algorithm with Brute-Force Approach**

## 5. Experiments

In this chapter the experiments performed on the Hamiltonian cycle algorithms on various graphs are described. The types of graphs that are used to test the Hamiltonian cycle algorithms are specified in Section 5.1. The experimental methodology used is outlined in Section 5.2. The results of the experiments performed on the three algorithms used are discussed in Section 5.3. The summary of the experiments is given in Section 5.4.

### 5.1 Input Graphs

The graphs that are used to test the algorithms discussed in the earlier chapter are 4-regular planar graphs that are 2-connected and 4-edge connected.

In this thesis two types of 4-regular planar graphs are used to test the Hamiltonian cycle algorithms. Algorithms to randomly generate both types of graphs were developed by Diao, Ernst, and Ziegler and are described in [4]. One type of graph is generated such that all the graphs always contain a Hamiltonian cycle. In this thesis these graphs are referred to as Ham graphs. Proper care has been taken in embedding the Hamiltonian cycle in the Ham graphs to make sure none of the algorithms have an advantage. The other type of graph is referred to as Tree graphs, and no prior information about the existence of a Hamiltonian cycle is known for each individual graph.

The Ham graphs are used to test the algorithms' ability to find Hamiltonian cycles. Since the algorithms might not find a Hamiltonian cycle, this allows us to know whether it is due to the algorithm (always in the case of a Ham graph) or due to the



graph not having a Hamiltonian cycle. Multiple edges and loop edges of the graph are removed before executing the algorithms.

The purpose of the experiments in the thesis is to investigate the algorithms' performance on 4-regular planar graphs.

## **5.2 Experimental Methodology**

In this section the experimental methodology for various algorithms used in the thesis is described.

Experiments were performed on the graphs of 20 to 550 vertices in this thesis. In our experiments three different sets of 20 randomly generated graphs are taken at each data point; the algorithms are executed and the results were averaged and reported.

A time limit of 600 seconds (i.e., 10 minutes) is used in Vandegriend backtrack algorithm and in the backtrack algorithm using the brute-force approach. As the Vandegriend backtrack algorithm picks the initial vertex randomly in the search, the results differ for various executions, so this algorithm is executed 10 times on each graph and the results are averaged.

Experiments were performed on several identical computers, each of which are Intel Pentium 4, 2.8 GHz CPU with 512 MB RAM and Windows XP 2002 Service Pack 2 operating system.

## **5.3. Results**

In this section the results of various experiments performed on Backtrack algorithm using brute-force approach, Vandegriend backtrack algorithm and SemiHam heuristic algorithm, for Ham graphs and Tree graphs, are shown and discussed.

### 5.3.1 Backtrack Algorithm using Brute-Force Approach

Table 2 shows the results of the average number of Ham graphs and Tree graphs in which the backtrack algorithm using brute-force approach found a Hamiltonian cycle. The graph shown in Figure 25 illustrates the percentage of graphs in which this algorithm found a Hamiltonian cycle. The graphs shown in Figure 26 illustrate the time taken by this algorithm to find Hamiltonian cycle in Ham graphs and in Tree graphs. These figures show that the algorithm takes more time to find the Hamiltonian cycle as the size of the graph increases. When the size reaches more than 80 vertices in Ham graphs and 60 vertices in Tree graphs the algorithm is unable to find the Hamiltonian cycle in 90% of the graphs, and so it loses its ability to find a Hamiltonian cycle.

The graph shown in Figure 25 illustrates that the algorithm can find a Hamiltonian cycle in more Ham graphs than in Tree graphs. The algorithm loses its ability to find a Hamiltonian cycle within the time limit of 600 seconds, as the size of the graph increases. This is due to the fact that this algorithm tries every vertex in the graph in each step, i.e., using the brute-force approach, as the size increases the algorithm requires more time to find the Hamiltonian cycle.

**Table 2. Results of Backtrack Algorithm using Brute-Force Approach**

Size of the Graph (N)	Average Number of Ham Graphs for which the algorithm found a Hamiltonian cycle (out of 20)	Average Number of Tree Graphs for which the algorithm found a Hamiltonian cycle (out of 20)
20 – 29	20	20
30 – 39	20	20
40 – 49	19.33	18.75
50 – 59	17.33	5
60 – 69	4.67	0.5
70 – 79	2.33	0
80 – 89	0	0
90 – 99	0	0

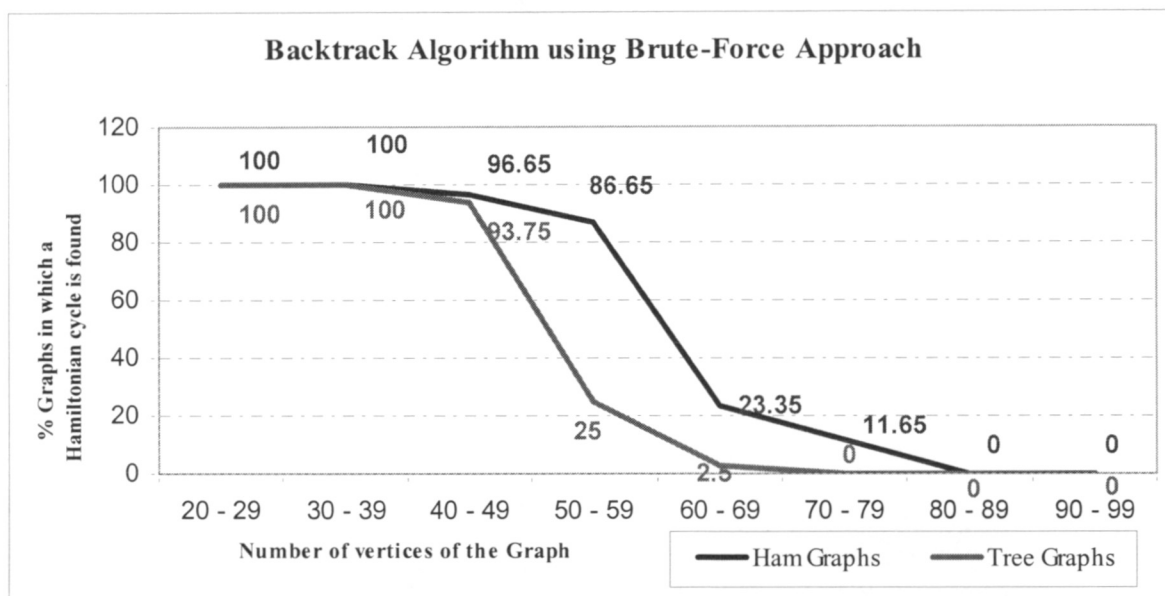


Figure 25: Illustrating the % Graphs Backtrack Algorithm using Brute-Force Approach found Hamiltonian Cycle

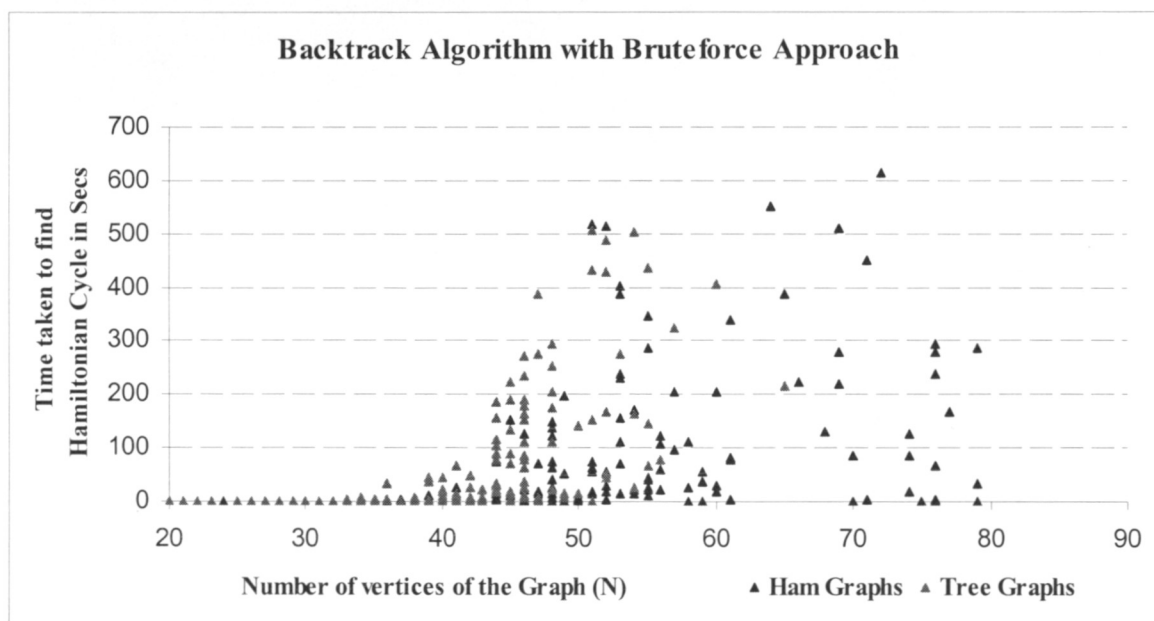


Figure 26: Time taken by Backtrack Algorithm using Brute-Force Approach to find Hamiltonian cycle in the Ham Graphs and Tree Graphs in Secs

### 5.3.2 Vandegriend Backtrack Algorithm

Table 3 shows the results of the average number of Ham graphs and Tree graphs for which the Vandegriend backtrack algorithm found a Hamiltonian cycle. The graph shown in Figure 27 illustrates the percentage of graphs in which this algorithm found a Hamiltonian cycle. The graph shown in Figure 28 illustrates the time taken by this algorithm to find Hamiltonian cycle in Ham graphs, and in Tree graphs. The times shown in Figure 28 are the average time taken by this algorithm to solve the same graph ten times as the Vandegriend backtrack algorithm involves randomness and results differ when executed different times. But the times near 600 seconds are the average time the algorithm found a Hamiltonian cycle before the timer has expired.

These graphs show that the algorithm takes more time to find the Hamiltonian cycle, as the size of the graph increases. When the size reaches more than 190 vertices in Ham graphs, and 150 vertices in Tree graphs, the algorithm is unable to find the Hamiltonian cycle in 90% of the graphs. Therefore it loses its ability to find a Hamiltonian cycle.

The graph shown in Figure 27 illustrates that the algorithm can find a Hamiltonian cycle in more number of Ham graphs than in Tree graphs. The algorithm loses its ability to find a Hamiltonian cycle within the time limit of 600 seconds, as the size of the graph increases.

Table 3: Results of Vandegriend Backtrack Algorithm

Size of the Graph (N)	Average Number of Ham graphs for which the algorithm found a Hamiltonian Cycle (out of 20)	Average Number of Tree maps for which the algorithm found a Hamiltonian Cycle (out of 20)
20 – 29	20	20
30 – 39	20	20
40 – 49	20	20
50 – 59	20	20
60 – 69	20	20
70 – 79	20	20
80 – 89	20	19.66
90 – 99	20	18.93
100 – 109	19.81	16.38
110 – 119	19.56	12
120 – 129	18.19	9.52
130 – 139	16.67	3.83
140 – 149	9.89	1.97
150 – 159	9.04	0.97
160 – 169	3.96	0.69
170 – 179	2.85	0.34
180 – 189	2.37	0.07
190 – 199	1.87	0
200 – 209	1.62	0
210 – 219	1.31	0
220 – 229	0.58	0
230 – 239	0.11	0
240 – 249	0.11	0

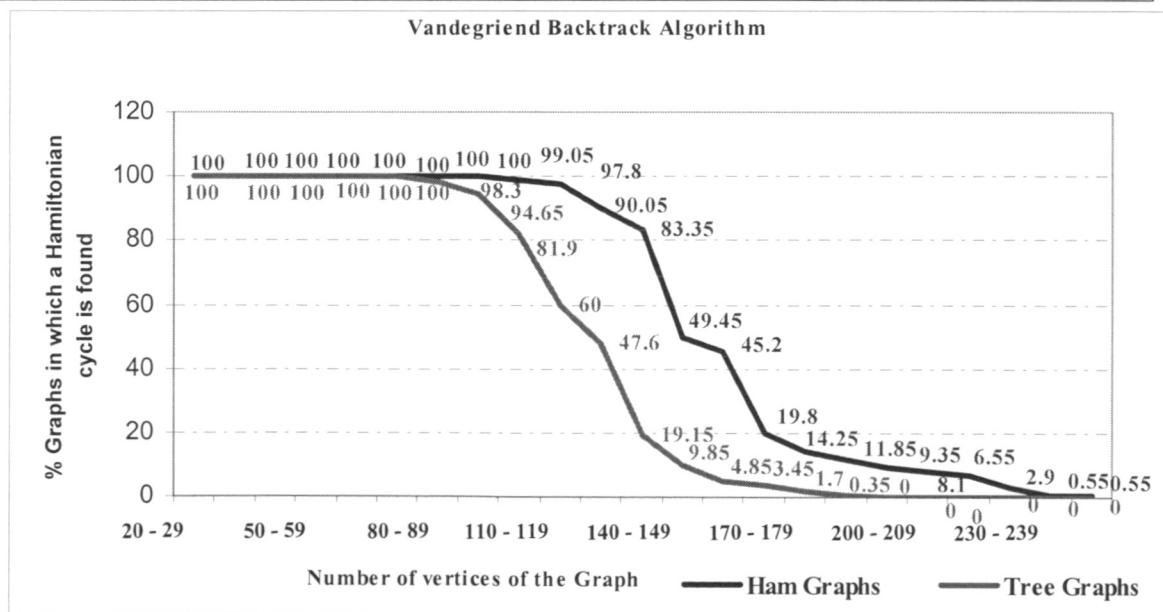


Figure 27: Illustrating the % Graphs Vandegriend Backtrack Algorithm found Hamiltonian Cycle

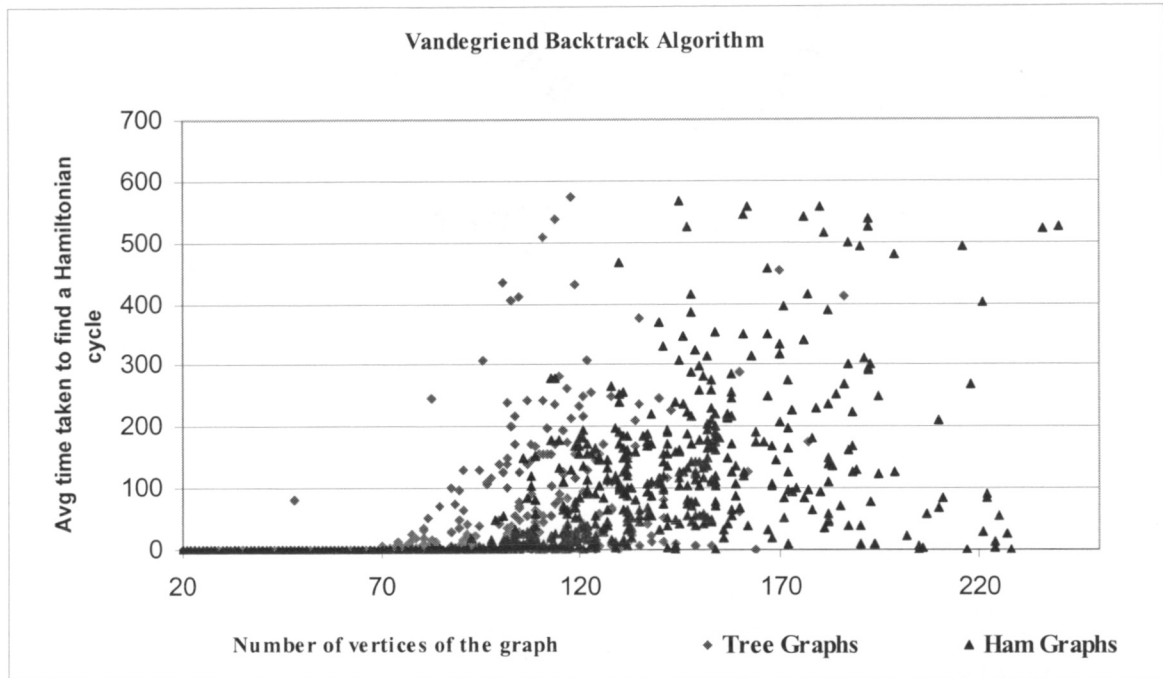


Figure 28: Average Time taken by Vandegriend Backtrack Algorithm to find Hamiltonian cycle in Ham Graphs and Tree Graphs

### 5.3.3 SemiHam Heuristic Algorithm

Table 4 shows the results of the average number of Ham graphs and Tree graphs, for which the SemiHam heuristic algorithm found a Hamiltonian cycle. The graph shown in Figure 29 illustrates the percentage of graphs in which the algorithm found a Hamiltonian cycle. The graph shown in Figure 30 illustrates the time taken by this algorithm to find a Hamiltonian cycle, in Ham graphs and in Tree graphs. When the size reaches more than 350 vertices in Ham graphs, and 550 vertices in Tree graphs, the algorithm is unable to find the Hamiltonian cycle in 90% of the graphs. Therefore it loses its ability to find a Hamiltonian cycle. This is due to the limitation of the heuristic algorithm for 4-regular planar graphs or even due to the limit on the number of rotated paths the algorithm can investigate.

The graph shown in Figure 29 illustrates that the algorithm can find a Hamiltonian cycle in more Tree graphs than in Ham graphs. This result differs from the previous two algorithms.

### 5.3.4 All the three Algorithms

The graphs shown in Figure 31 and 32 illustrate the time taken to find a Hamiltonian cycle, by all the three algorithms in Ham graphs and in Tree graphs, in milliseconds. In these graphs the line BABF indicates the time taken to find a Hamiltonian cycle by Backtrack algorithm using brute-force approach, VBA indicates Vandegriend backtrack algorithm, and SHA indicates SemiHam heuristic algorithm. In these graphs the Backtrack algorithm with brute-force approach takes more time to find a Hamiltonian cycle than the other two methods, as it tries all the vertices of the graph in every step, i.e., the brute-force approach.

**Table 4: Results of SemiHam Heuristic Algorithm**

Size of the Graph (N)	Average Number of Ham graphs in which the algorithm found a Hamiltonian Cycle (out of 20)	Average Number of Tree graphs in which the Algorithm found a Hamiltonian Cycle (out of 20)
20 – 49	19.56	19.76
50 – 99	17.29	18.48
100 – 149	13.79	15.58
150 – 199	8.91	12.94
200 – 249	5.74	11.55
250 – 299	3.17	9.7
300 – 349	1.65	8.18
350 – 399	1.72	7
400 – 449	0.59	4.95
450 – 499	0.33	3.17
500 – 559	0	2.44

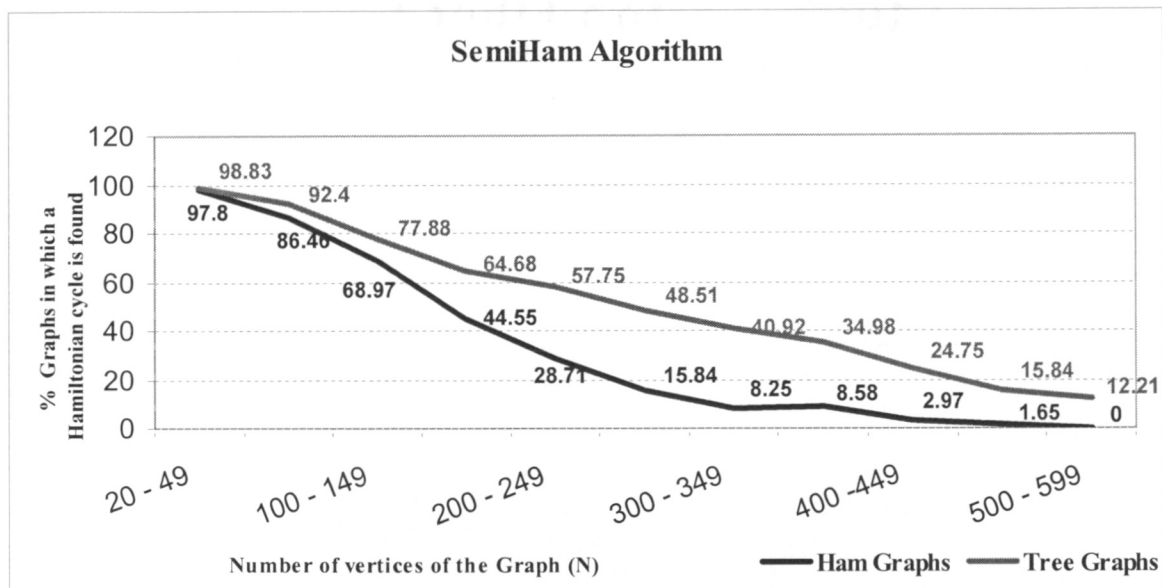


Figure 29: Illustrating the % Graphs found Hamiltonian by SemiHam heuristic Algorithm

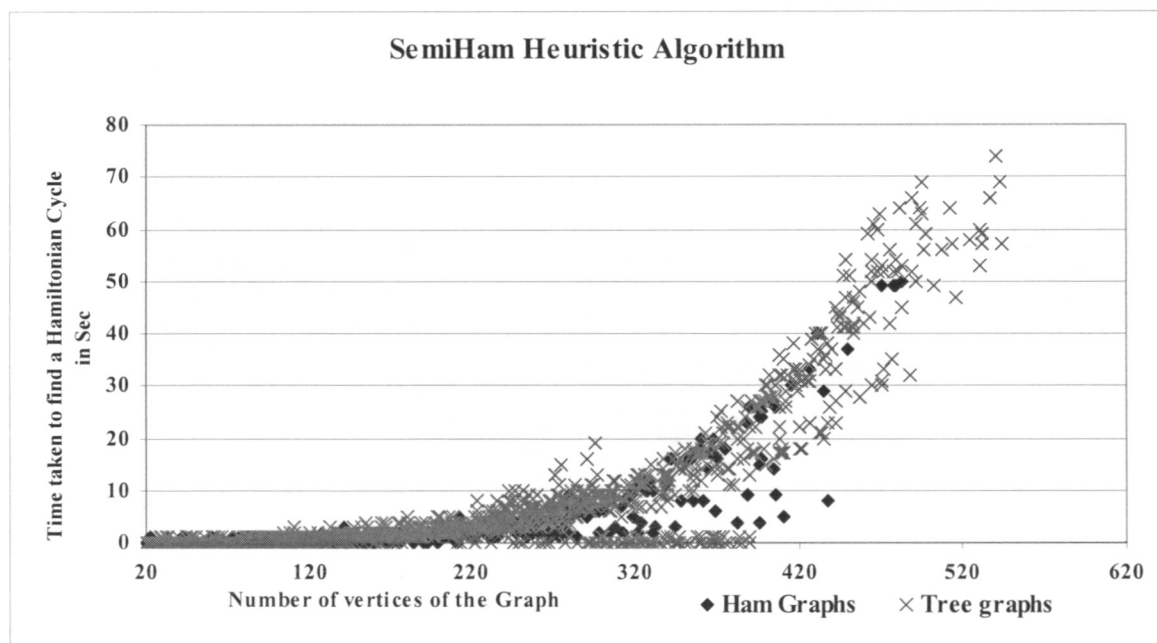


Figure 30: Time taken by SemiHam Algorithm to find Hamiltonian cycle in Ham Graphs and Tree 3Graphs



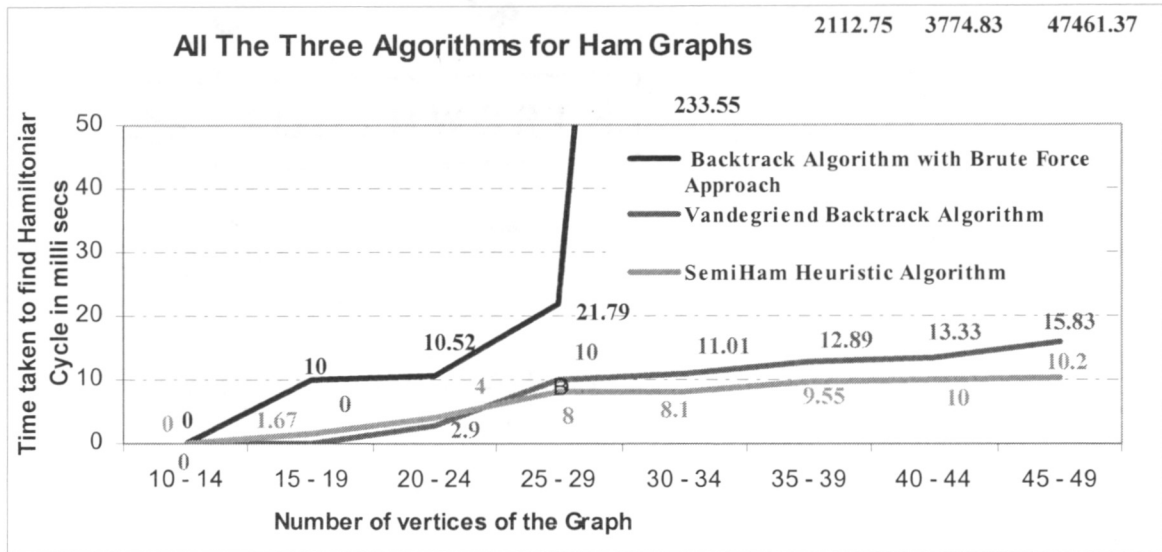


Figure 31: Illustrating the time taken to find Hamiltonian cycle by all three algorithms for Ham Graphs

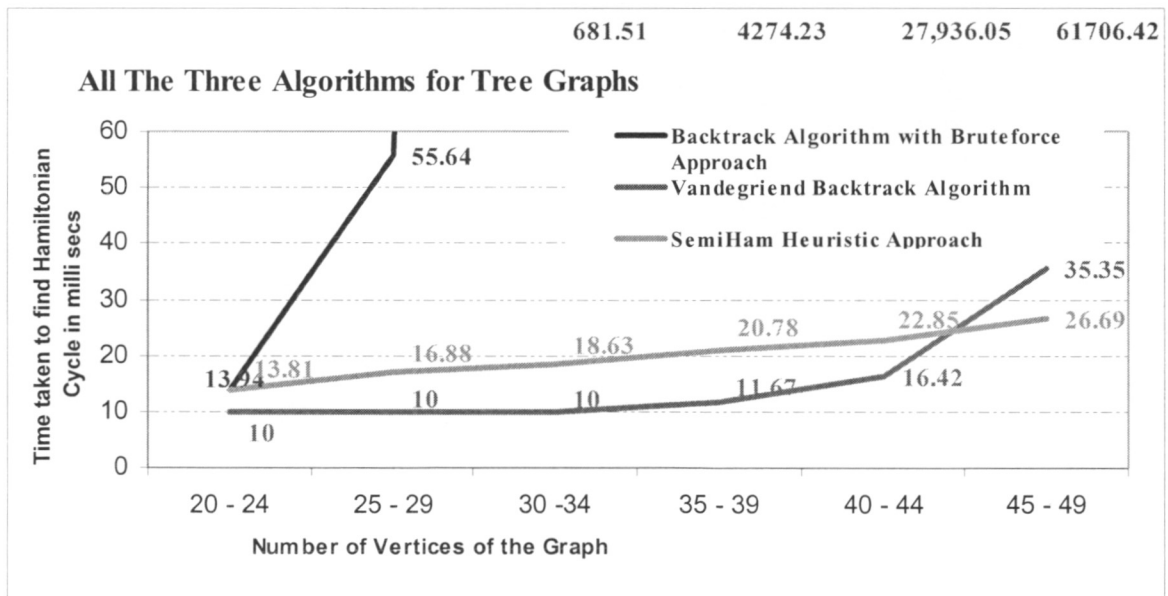


Figure 32: Illustrating the time taken to find Hamiltonian cycle by all three algorithms for Tree Graphs

#### 5.4. Summary of Experiments

The SemiHam heuristic algorithm is better suited for both Tree graphs and Ham graphs. It finds a Hamiltonian cycle in 4-regular planar graphs tested of size up to 550. The SemiHam heuristic algorithm does not test all possible conditions to find a solution,

so if this method returns that the given graph is non-Hamiltonian, then it can be Hamiltonian or non-Hamiltonian. This algorithm finds a Hamiltonian cycle in more Tree graphs than in Ham graphs.

Vandegriend backtrack algorithm found a Hamiltonian cycle in 4-regular planar graphs of size up to 250 within a time limit of 600 seconds. The backtrack algorithm using brute-force approach found a Hamiltonian cycle in 4-regular planar graphs of size up to 80, within a time limit of 600 seconds. The SemiHam heuristic algorithm is able to find a Hamiltonian cycle in a higher percentage of 4-regular planar graphs than the other two algorithms.

## 6. Conclusions

In this thesis, various Hamiltonian cycle algorithms, the basic ideas and their implementation, are studied and tested with two types of randomly generated 4-regular planar graphs that are 2-connected and 4-edge connected, i.e., Ham graphs and Tree graphs. The performance of these algorithms on Ham graphs and Tree graphs are examined in this thesis.

Among the three methods used, the SemiHam heuristic algorithm finds the Hamiltonian cycle in less time, and in higher % of graphs. It finds a Hamiltonian cycle in at least 10% of Tree graphs up to 550 vertices, and Ham graphs up to 350 vertices.

Of the three algorithms used, Backtrack algorithm with the brute-force approach takes the most time to find a Hamiltonian cycle in a graph (see Figures 31 and 32). So this algorithm cannot find a Hamiltonian cycle in the Ham graphs with more than 80 vertices, and Tree graphs with more than 60 vertices, within a time limit of 600 seconds in more than 10% of the graphs given.

The Vandegriend backtrack algorithm finds a Hamiltonian cycle in the Ham graphs with more than 190 vertices, and in Tree graphs with more than 140 vertices, within a time limit of 600 seconds in more than 10% of the graphs given.

In this thesis we have found that the SemiHam heuristic algorithm is the most efficient one among the three to find a Hamiltonian cycle, in 4-regular planar graphs. It finds a Hamiltonian cycle in larger number of Tree graphs than Ham graphs.

## Bibliography

- [1] Bela Bollobas, Trevor I. Fenner and Alan M. Frieze. An algorithm for finding Hamilton paths and cycles in random graphs. *Combinatorica*, 7(4): 327-341, 1987.
- [2] John A. Bondy and U.S.R. Murthy. *Graph theory with Applications*. Elsevir, Amsterdam, 1976.
- [3] Norishige Chiba and Takao Nishizeki. The Hamiltonian Cycle Problem is Linear-Time Solvable for 4-Connected Planar Graphs. *Journal of Algorithms* 10, 187-211, 1989.
- [4] Yuanan Diao, Claus Ernst, Uta Ziegler. Generating Large Random Knot Projections.
- [5] Jeremy Frank and Charles Martel. Phase transitions in the properties of random graphs. In *CP'95 Workshop: Studying and Solving Really Hard Problems*, pages 62- 69, September 1995.
- [6] Gregory Gutin, *Polynomial algorithms for finding paths and cycles in quasi-transitive digraphs*. Tel Aviv University, Israel, March 9, 2003.  
<http://www.cs.rhul.ac.uk/home/gutin/paperstsp/polya2.pdf>
- [7] Richard Johnsonbaugh, *Backtrack algorithm to find Hamiltonian Cycle using Brute-force Approach*. DePaul University, Chicago.  
<http://condor.depaul.edu/~rjohnson/source/hamilton.c>
- [8] Eran Keydar. *Finding Hamiltonian cycles in semi-random graphs*. Masters thesis, Weizmann Institute of Science, Rehovot, Israel, 2002.  
<http://www.wisdom.weizmann.ac.il/~erank/thesis/thesis/main.ps>
- [9] Rashid Bin Muhamed, *Traveling Salesman Problem*.  
<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/AproxAlg or/TSP/tsp.htm>
- [10] Gabriel Nivasch. *Experimental Results on Hamiltonian cycle finding algorithms*. May 2003.  
<http://www.wisdom.weizmann.ac.il/~gabrieln/papers/HamiltonReport.pdf>
- [11] William. T. Tutte, A theorem on planar graphs, *Trans Amer. Math. Soc.* 82 (1956), 99-116.

- [12] William. T. Tutte, Bridges and Hamiltonian circuits in planar graphs, *Aequationes Math.* 15 (1977), 1-33.
- [13] Basil Vandegriend. *Finding Hamiltonian cycles: Algorithms, graphs and performance*. Masters thesis, University of Alberta, Edmonton, Canada, Feb 1998. <http://web.cs.ualberta.ca/~joe/Theses/vandegriend.html>
- [14] All-to-all broadcast algorithms in SF networks,  
<http://www.cs.wisc.edu/~tvrdik/13/html/Section13.html>
- [15] Lecture notes on part II of introduction to cryptography,  
<http://www.beznosov.net/konstantin/doc/talks/cot6421-handouts.pdf>
- [16] MathWorld, <http://mathworld.wolfram.com>
- [17] The Hamiltonian Cycle Problem,  
<http://www.math.usf.edu/~jonoska/bio-comp/node3.html>